精译版

勒索软件利用 NSIS 安装程序安装

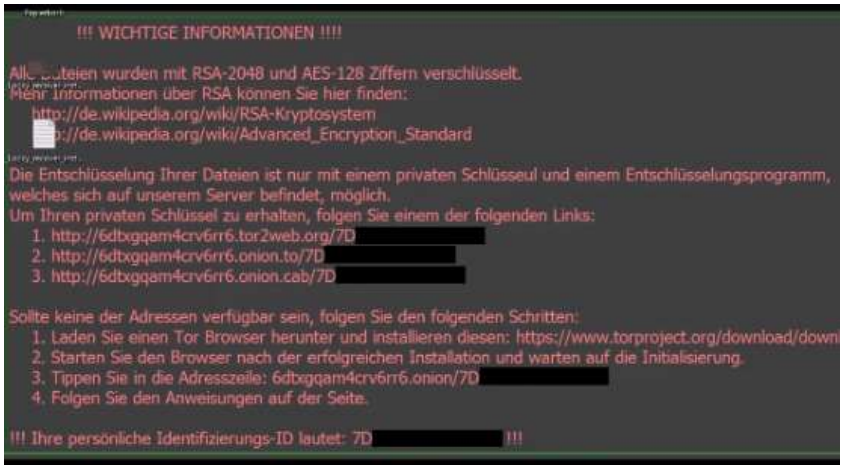
非官方中文译文·安天技术公益翻译组 译注

文档信息			
原文名称	Ransomware Installation Using NSIS Installer		
原文作者	Tom Nipravsky	原文发布日期	2017 年 3 月 20 日
作者简介	Tom Nipravsky 是 Deep Instinct 的安全研究员。https://www.linkedin.com/in/tom-nipravsky-7b7aa021/		
原文发布单位	Deep Instinct		
原文出处	http://blog.deepinstinct.com/2017/03/20/ransomware-installation-method-using-nsis-installer/		
译者	安天技术公益翻译组	校对者	安天技术公益翻译组
免责声明	• 本译文译者为安天实验室工程师，本文系出自个人兴趣在业余时间所译，本文原文来自互联网的公共方式，译者力图忠于所获得之电子版本进行翻译，但受翻译水平和技术水平所限，不能完全保证译文完全与原文含义一致，同时对所获得原文是否存在臆造、或者是否与其原始版本一致未进行可靠性验证和评价。 • 本译文对应原文所有观点亦不受本译文中任何打字、排版、印刷或翻译错误的影响。译者与安天实验室不对译文及原文中包含或引用的信息的真实性、准确性、可靠性、或完整性提供任何明示或暗示的保证。译者与安天实验室亦对原文和译文的任何内容不承担任何责任。翻译本文的行为不代表译者和安天实验室对原文立场持有任何立场和态度。 • 译者与安天实验室均与原作者与原始发布者没有联系，亦未获得相关的版权授权，鉴于译者及安天实验室出于学习参考之目的翻译本文，而无出版、发售译文等任何商业利益意图，因此亦不对任何可能因此导致的版权问题承担责任。 • 本文为安天内部参考文献，主要用于安天实验室内部进行外语和技术学习使用，亦向中国大陆境内的网络安全领域的研究人士进行有限分享。望尊重译者的劳动和意愿，不得以任何方式修改本译文。译者和安天实验室并未授权任何人士和第三方二次分享本译文，因此第三方对本译文的全部或者部分所做的分享、传播、报道、张贴行为，及所带来的后果与译者和安天实验室无关。本译文亦不得用于任何商业目的，基于上述问题产生的法律责任，译者与安天实验室一律不予承担。		

勒索软件利用 NSIS 安装程序安装

Tom Nipravsky

2017 年 3 月 20 日



简介

过去几年，我们见过各种恶意代码执行方法。

最近，我们发现了一种非常复杂的方法，它使用多层入侵技术，利用 NSIS 安装程序，XOR 加密技术，代码注入技术，甚至使用 Heaven’ s Gate 技术来打包内部组件。目前，两个最著名的勒索软件在使用这一技术：Locky 和 Cerber，都是最新版本。

NSIS 还使用了一种“系统”插件技术，该技术允许 NSIS 安装程序调用 Win32 API（应用程序接口），并允许攻击者分配可执行内存和执行代码桩，然后代码桩解密勒索软件的载荷。这样，NSIS 就可以隐藏其内容，规避安全厂商的检测。事实是，内存中发生的所有恶意活动都会非常难以检测。

勒索软件也使用“Heaven’ s Gate”技术，主要有两个原因。

第一，该技术允许从 32 位的进程中调用 64 位的代码。勒索软件使用该技术可以绕过 ntdll.dll 的 API 钩子，直接使用系统调用而非标准的 API（后者只能通过 64 位代码调用）。这一技术被流行木马使用过多次（例如银行木马 Vawtrak，它是最著名的使用 Heaven’s Gate 技术的木马家族）。

第二，该技术在模糊代码方面效果很好。因为几乎所有的调试器从 32 位的进程中执行 64 位的代码效果都不太好（只有使用 windbg 进行远程内核调试时，才能实现这一点）。调试器之所以无法很好的处理此类情况，是因为其设计初衷是一次只处理一个架构。

更糟糕的是，勒索软件还使用了一种“进程挖空”（process hollowing）的技术来执行安装程序。该技术使得攻击者可以在暂停状态下创建新的进程，并用隐藏进程替代其映像。而安装程序本身也已在 NSIS 内部加密了，因此安全软件厂商无法追踪。勒索软件只在运行时才能被解密。

该技术的实施很有趣，它不是典型的“进程挖空”。勒索软件在两个进程之间创建一个共享区（内含勒索软件安装程序），然后在新区中执行新进程。

我们在研究过程中碰到了使用完全相同的规避技术的勒索软件：Locky 和 Cerber（不同的版本）都表现出完全相同的行为，先使用 NSIS，相同的 NSIS 脚本模糊处理方法，相同的 XOR 加密技术以及运行时解密安装程序，使用 Heaven’s Gate 技术，最后，执行相同的“进程挖空”技术。

NSIS 安装程序

该安装方法使用了一种技术——NSIS（Nullsoft Scriptable Install System，Nullsoft 脚本安装系统）安装程序。NSIS 是一个专业的开源系统，用于创建 Windows 安装程序。与其它只能根据文件表和注册表项生成安装程序的系统不同，NSIS 有着强大的脚本语言。

这种脚本语言是专为安装程序设计的，它的命令可以执行多个安装任务。它可以轻易的添加逻辑，处理不同的更新，版本检测等等。

NSIS 最主要的特征是使用能够拓展 NSIS 功能的插件。这些插件可以使用 C，C++，Delphi 或其他语言编写，并用于执行安装任务或扩展安装程序接口。

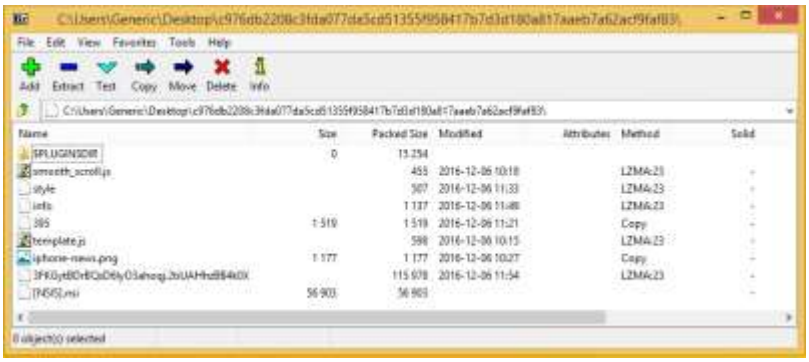
Cerber 和 Locky 使用了一个“系统”（System）插件，这允许安装程序与操作系统通信，甚至可以调用外部 DLL 导出的函数（因此，它允许安装程序调用内核 32.dll 导出的 Win32API）。Locky 和 Cerber 安装程序就是利用了这一优势。而且，通过调用 Win32 API，Cerber 和 Locky 可以执行恶意 shellcode。

在本报告中，我们分析了 Locky 勒索软件（SHA256: c976db2208c3fda077da5cd51355f958417b7d3d180a817aaeb7a62acf9faf83）。但是请注意，这些结论也适用于 Locky 和 Cerber 的其它版本。

请参考本文末的感染信标。

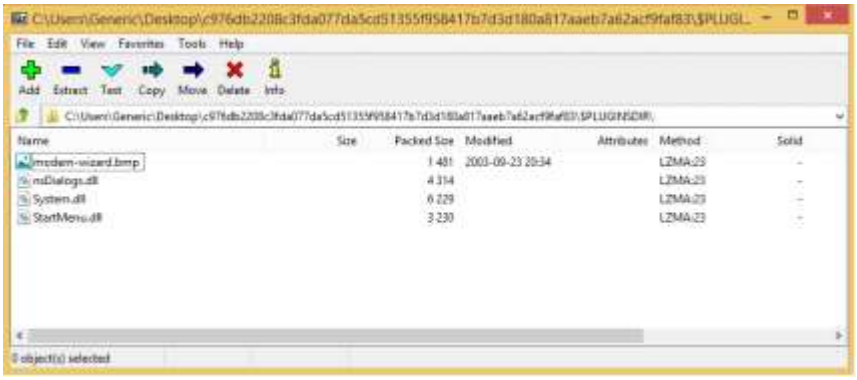
NSIS 安装程序结构

NSIS 安装程序可以使用 7-Zip 解压。而 Locky 的 NSIS 文件也非常简单，包含下列文件和目录：



- [NSIS].nsi：主要脚本文件（明文）。

- 3FKGytBDrBQsD6lyO3ahoqj.2tiUAHhzBB4k0X ：Locky 执行方法的实际 shellcode 和“运作逻辑”。该文件还包含加密内容（本文随后会深入解释）。
- PLUGINSDIR：该目录包含安装程序使用的插件。



脚本分析

与之前提到的一样，NSIS 是一个基于脚本的安装程序。通过使用 7-Zip 解压，我们能够打开安装程序并提取文件，包括真正用于创建安装程序的实际脚本。

查看[NSIS].nsi 文件后，我们分析了安装程序背后的逻辑，并能看到它正在做什么。

根据 NSIS 的文件记录，“.onInit”函数是一个回调函数。当安装程序快完成初始设置时，进行调用，因为，它是安装程序真正的“主要”（main）函数。

在查看 Locky 安装程序上的“.onInit”后，我们发现了一个执行 shellcode 的简单逻辑。

首先，它在%temp%目录下（安装目录）写入安装文件，然后利用“系统”（System）插件（调用“CreateFile”函数），打开一个到包含实际 shellcode 的文件句柄。

脚本作者之所以这样编写的原因可能是为了模糊和混淆代码。

根据 NSIS 文件记录，函数的输出结果（本例中——句柄）可以通过输入“r0”变量获得。

第二步，使用带有 PAGE_EXECUTE_READWRITE 权限的（0x40）的“VirtualAlloc”函数分配内存区域。

```
StrCpy $R4 kerne
StrCpy $R4 $R4132::VirtualAlloc
StrCpy $R4 $R4(
StrCpy $R4 "$R4i 0"
StrCpy $R4 "$R4, i 120464, "
StrCpy $R4 "$R4i 0x3000, i 0"
StrCpy $R4 "$R4x40) p"
StrCpy $R4 "$R4 .r1"
System::Call $R4
```

分配区域的大小与包含 shellcode 的文件和加密内容的大小一样（3FKGytBDrBQsD6lyO3ahoqj.2tiUAHhzBB4k0X），恰好是 120464 个字节。新分配的内存地址放在“r1”变量内。

第三步，读取“3FKGytBDrBQsD6lyO3ahoqj.2tiUAHhzBB4k0X”文件的内容（“r0”句柄从“CreateFile”函数返回）放到新分配的内存区域内（“r1”变量从“VirtualAlloc”函数返回）。

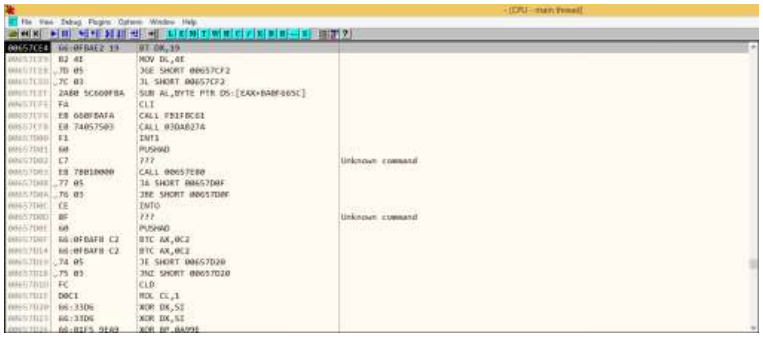
```
StrCpy $R5 kernel32:
StrCpy $R5 $R5:ReadFile
StrCpy $R5 "$R5(i r"
StrCpy $R5 $R50
StrCpy $R5 "$R5, p r1"
StrCpy $R5 "$R5, i 120464, "
StrCpy $R5 $R5t.,)
System::Call $R5
```

既然代码已经就位，第四步就是调用 shellcode 的主函数并把“\3FKGytBDrBQsD6lyO3ahoqj.2tiUAHhzBB4k0X”作为一个参数。主要的 shellcode 函数开始与分配的内存 97508 建立联系。

```
IntOp $1 $1 + 97508
StrCpy $R2 :
StrCpy $R2 $R2
StrCpy $R2 $R2:$1(t
StrCpy $R2 "$R2 '"
StrCpy $R2 $R2\3FKGytBDrBQsD6lyO3ahoqj.2tiUAHhzBB4k0X
StrCpy $R2 $R2
StrCpy $R2 $R2)
System::Call $R2
```

Shellcode

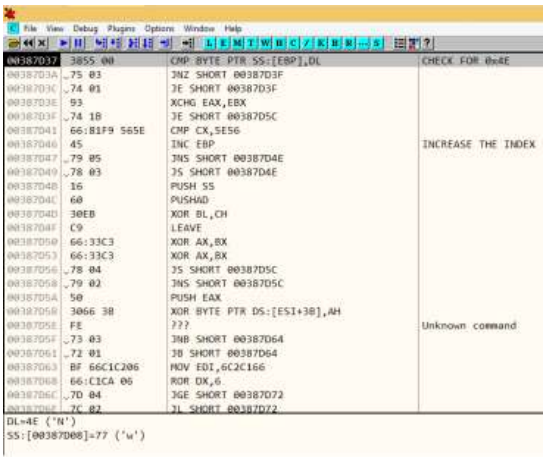
我们可以清楚的看到 shellcode 以某种方式模糊处理了。



穿过该代码，我们发现代码 XOR 的方式，非常有趣。shellcode 的工作原理是每一部分掌管代码 XOR 的下一个部分，然后加壳代码的下一部分掌管者 XOR 的再下一个部分依次类推。在具体的实例中，代码 XOR 花了 2 个周期才进入有效的“运作逻辑”执行阶段，但也有其它例子长短不一，有些花了 7 个周期，有的花了 5 个周期。

正在异或处理的进程

这个程序的工作原理很有趣。首先，它穿过 XOR 的内容，然后查找“0x4E”字节。



因为代码已 XOR 过了，所以 OllyDbg 无法解析整段代码以及丢失的一些信息。一旦代码增加一个指数（EBP 寄存器），就跳回对 0x4E 进行有效检测。

```
00387D30 77 05          JA SHORT 00387D37
```

代码一发现包含 0x4E 的地址，立马就搜索 XOR 密匙。

shellcode 搜索 XOR 密匙的方法非常简单，但有一点复杂。0x4E 是下一段将要“DE-XOR”代码的标记。每一段都有如下的数据头：

Marker (0x4E) – 1 byte	Section Size – 4 bytes	XOR keys – 4 bytes Key1 | Key2 | Key3 | Key4 The keys need to be initialized	Value – 4 bytes	Code to be De-XOR. The size is “Section Size”

shellcode 初始设置 XOR 密匙的方法很简单。首先，它设定 ESI 寄存器的初始值为 0，然后检查 Value ^ ESI 的结果是否与“XOR 密匙”内部值匹配。一旦找到匹配值，加壳代码就利用 ESI 替代“XOR 密匙”的内容，然后密匙也已完成了初始设置。

```
00387DA4  3B45 04       CMP EAX,DWORD PTR SS:[EBP+4]
00387DA7  74 16         JE SHORT 00387DBF
00387DA9  66:81FD 31CC  CMP BP,0CC31
00387DAE  46            INC ESI
00387DAF  EB C9         JMP SHORT 00387D7A          JUMP TO XOR ROUTINE
```

EAX 包含 Value ^ ESI 的输出结果，而[EBP+4]是未初始设置的 XOR 密匙。如果没有找到匹配值，ESI 增加一个值，然后 XOR 继续匹配，再次对比。

```
00387D92  8B45 08       MOV EAX,DWORD PTR SS:[EBP+8]
00387D95  66:C1CA 15    ROR DX,15
00387D99  66:C1C2 15    ROL DX,15
00387D9D  33C6          XOR EAX,ESI
00387D9F  7E 03         JLE SHORT 00387DA4          JUMP BACK TO COMPARISON
```

一旦 ESI 给定的值对比正确之后，就用 ESI 替代“XOR 密匙”。

如果，例如 ESI=0xABCDEF12 给定的输出结果正确，那么 XOR 密匙将如下：

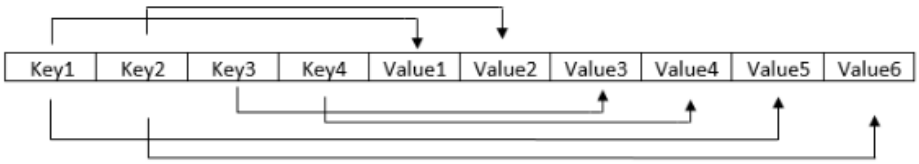
Key1 = AB

Key2 = CD

Key3 = EF

Key4 = 12

既然我们有了 XOR 密匙，分别从“Code to be DE-XOR”中循环“Section Size”的次数和用密匙 XOR 每个字节。



EDX 是计数器，我们可以检查没有通过的“Section Size”。

```
00387E5E  3B55 00       CMP EDX,DWORD PTR SS:[EBP]
00387E61  ^7C 98        JL SHORT 00387DFB           JUMP TO XOR THE NEXT BYTE IN LINE
```

循环继续，加壳代码 XOR 排队等待下一个字节。

CL 寄存器包含当前用于 XOR 活动的密匙。如果计数器到了 4，EBX 需要设为 0 然后再次循环。

```
00387E1B  304C2A 08     XOR BYTE PTR DS:[EDX+EBP+8],CL
00387E1F  43            INC EBX
00387E20  42            INC EDX
00387E21  83FB 04       CMP EBX,4
00387E24  75 05         JNZ SHORT 00387E2B
00387E26  74 03         JE SHORT 00387E2B          JUMP AND RESET THE COUNTER
00387E28  FE            ???                        Unknown command
00387E29  E3 9A         JECXZ SHORT 00387DC5
00387E2B  7C 10         JL SHORT 00387E3D
00387E2D  7A 04         JPE SHORT 00387E33
00387E2F  78 02         JPO SHORT 00387E33
00387E31  75 AB         JNZ SHORT 00387DDE
00387E33  66:3BF8       CMP DI,AX
00387E36  33DB          XOR EBX,EBX                RESET THE COUNTER
```

一旦 shellcode DE-XOR 了所有相关的字节，就会跳到代码的下一部分。

```
00387E6B  66:2BFD       SUB DI,BP
00387E6E  83C5 08       ADD EBP,8
00387E71  FFE5          JMP EBP
```

整个进程再重复一次，然后跳到 Locky 真正的运作逻辑。

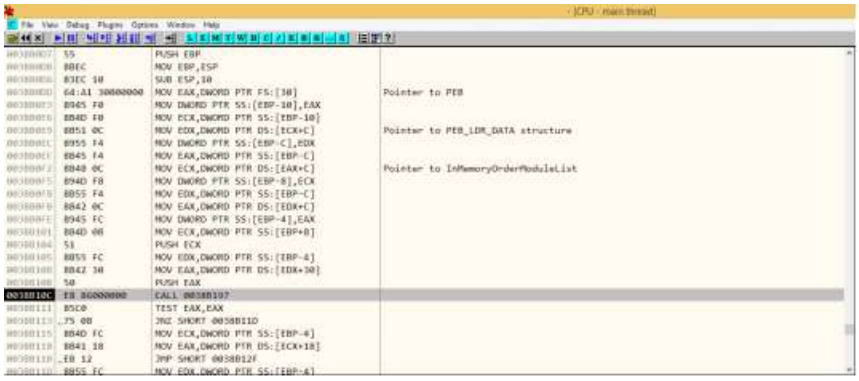
运作逻辑

Locky 和 Cerber 利用著名的挖空进程技术（有很多执行技术）进行安装。在深入探究该代码后，我们发现该攻击的执行方法非常有趣。

——获取内核 132&NTDLL 映像库：

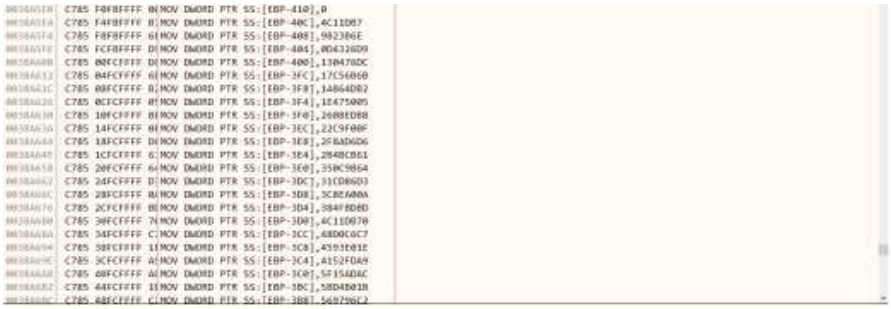
Locky 和 Cerber 使用著名的技术查找核心库的基地址。

因为 shellcode 需要调用 Win32API，所以需要查找其地址。为此，shellcode 访问进程环境块，里面包含很多的加载模块链接和基地址。具体接收的函数作为库名输入值和请求库映像库返回。



代码一获取内核 32.dll 映像库地址，就穿过导出表并查找内核 32 的 22 函数，advapi32.dll 的 10 函数和 ntdll.dll 的 4.

代码查找具体函数地址的方法非常特别。它绕过 CRC32 计算字符串的搜索函数，经过具体的 DLL 导出表并计算每一个导出函数的 CRC32。一旦匹配到查找的导出函数，shellcode 检索地址并存储。（因为我没有细查代码如何计算 CRC32，但似乎是因为使用了 CRC32“0x04C11DB7”多项式值查找表）。函数搜索的方法*可能*隐藏在它们真正搜索的函数中。



映射 NTDLL

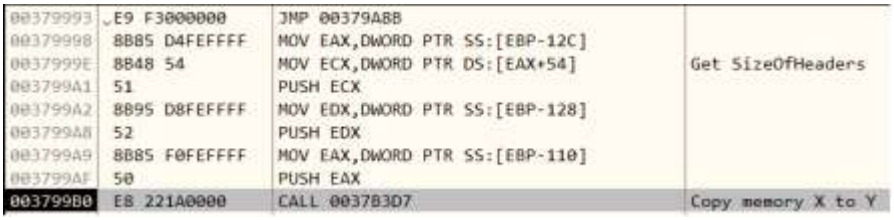
shellcode 内部发生了一件有趣的事，正在映射 ntdll.dll。

首先，代码打开了一个连到 ntdll.dll 的句柄，利用 VirtualAlloc 分配足够的空间（GetFileSize 接收到文件大小与 ntdll.dll 大小相同）。然后，代码读取 ntdll.dll 中内容到新分配内存中。请注意即使路径显示“C:\Windows\SYSTEM32”，实际是指“C:\Windows\SysWOW64”，因为该进程是 32 位的。

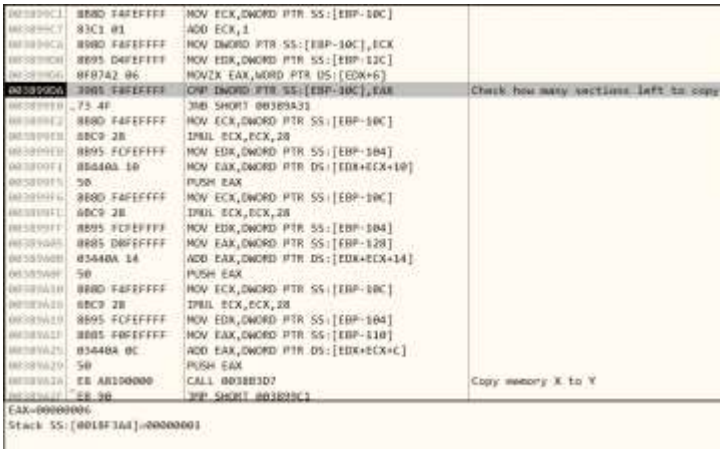
第二，代码跳过 IMAGE_OPTIONAL_HEADER，直接到 ntdll.dll 部分。

第三，给包含 ntdll 映射的图像分配足够的空间（与“SizeOfImage”的 ntdll 大小一致）。

第四，复制数据头（数据头的大小由 IMAGE_OPTIONAL_HEADER 内部的“SizeOfHeaders”确定）。

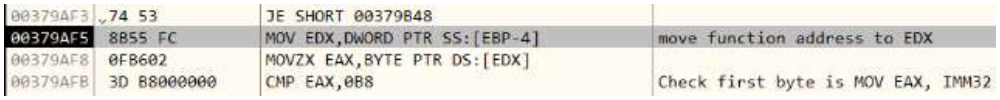


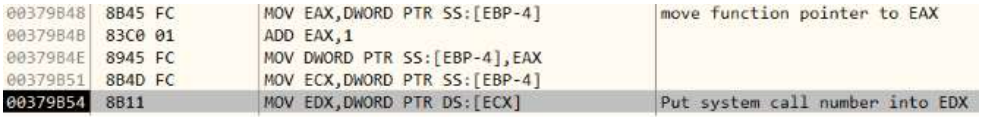
第五，复制部分代码（复制的部分数字是由 IMAGE_FILE_HEADER 内的“NumberOfSections”字段确定的）。



NTDLL 一映射到内存中，会发生一些很有趣的事情。代码会在映射 ntdll 中搜索具体的函数地址（使用上述 CRC32 方法）。一旦 NTDLL 获取了函数地址，shellcode 提取属于函数的系统调用标识符。要理解这一过程，我们需要理解系统调用数字是如何设定的：

每一个 Nt*函数以相同的操作码开头（MOV EAX, IMM32），接下来的四个字节决定着系统调用号。因此，我们希望第一个操作码是 B8h (MOV EAX, IMM32)开头的，然后接下来四个是系统调用号。这与 shellcode 上发生的一切完全一样：





这样做很可能是为了绕过安全解决方案的挂钩机制，从而规避被检测和获取基本的模糊代码。这一过程非常的简单——它不使用需要设置的正常高级别的 API，寄存器和调用 sysenter / syscall（系统输入- x32 架构，系统调用-x64 架构）。

系统调用的问题是不同 Windows 版本差别很大，而且也没有 API 可以获取系统调用号。但可以在运行时，使用 ntdll.dll 获取。让我们深入探究使用提取调用号到底会发生什么。

——Heaven’s Gate

shellcode 检索到系统调用号后，使用“Heaven’s Gate”技术从 32 位的进程中执行 64 位的代码。

Windows64 的位计算机优先执行 64 位的程序（64 位或 32 位），并优先执行 64 位的 ntdll 代码，因为 ntdll 负责进程初始化（64 位，甚至 32 位的程序）。直到后来 WoW64 接管了任务，载入 32 位版本的 ntdll.dll 并开始通过远程跳转到一个可计算的代码段执行。WoW64 只能在系统调用的情况下才能回调到 64 位的计算机内。载入的 32 位 ntdll.dll 包含一系列跳转到 64 位模式（而不是存在于 64 位 ntdll.dll 的 SYSCALL 指令）的指令，这样才可发布 SYSCALL 指令。

大致来说，WOW64 是由一整套 32 位的存根库组成，可以使得应用在 32 位与 64 位之间切换的同时，流畅运行。

32 位与 64 位代码之间转换，Heaven’ s Gate 技术实在是太简单了。因为 64 位的 Windows 上运行的每一个程序，都会分配两个代码段。

代码段 0x23 -> x86 模式

代码段 0x33 -> x64 模式

这也是 shellcode 上使用的方法。加壳代码收到符合要求的系统调用号后，切换到 x64 模式执行 SYSCALL 指令。

```
0037A55A  6A 33          PUSH 33                          push 33h as segment
0037A55C  E8 00000000    CALL 0037A561
0037A561  830424 05      ADD DWORD PTR SS:[ESP],5         Set offset of 64-bit code
0037A565  CB             RETF                             Far return
0037A566  2B65 EC        SUB ESP,DWORD PTR SS:[EBP-14]
0037A569  FF75 B8        PUSH DWORD PTR SS:[EBP-48]
0037A56C  59             POP ECX
0037A56D  FF75 C8        PUSH DWORD PTR SS:[EBP-38]
0037A570  5A             POP EDX
0037A571  FF75 F0        PUSH DWORD PTR SS:[EBP-10]
0037A574  41             INC ECX
0037A575  58             POP EAX
0037A576  FF75 F8        PUSH DWORD PTR SS:[EBP-8]
0037A579  41             INC ECX
0037A57A  59             POP ECX
0037A57B  FF75 D8        PUSH DWORD PTR SS:[EBP-28]
0037A57E  5F             POP EDI
0037A57F  FF75 E0        PUSH DWORD PTR SS:[EBP-20]
0037A582  5E             POP ESI
Return to 0033:0037A566
```

64 位代码内部，我们可以看到 SYSCALL 指令。

```
0037A59C  8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]     move system call number to EAX
0037A59F  0F05           SYSCALL
```

shellcode 一般使用 SYSCALL 指令绕过挂钩机制，使得代码难以调试，因为几乎所有的调试程序在处理跳转时效果都不太好(只有使用 windbg 远程内核调试时 ,才可以实现)。

而且，调试程序也不能很好的处理这类情况，因为调试程序是专为一次处理一个架构设计的。

——进程挖空技术

shellcode 试图利用 Heaven’ s Gate 实现多个目的：

——解密进程主机内部用作代码的 PE

还记得 NSIS 编写的临时文件“3FKGytBDrBQsD6lyO3ahoqj.2tiUAHhzBB4k0X”吗？是时候用它打开文件句柄了。

有了句柄后，NSIS 分配空间并读取新分配内存中所有文件的内容。然后，分配另外的存储区域，再在 offset 0x5C 中提取“3FKGytBDrBQsD6lyO3ahoqj.2tiUAHhzBB4k0X”。

分配完成后，NSIS 把“3FKGytBDrBQsD6lyO3ahoqj.2tiUAHhzBB4k0X” 0x64 中的内容复制到新分配的内存中（复制内容的大小与之前提取的大小一样）。获取了内容后，我们就开始解密了。

```
0037B572  68 000000F0      PUSH F0000000
0037B577  6A 18            PUSH 18
0037B579  6A 00            PUSH 0
0037B57B  6A 00            PUSH 0
0037B57D  8D8D 00FFFFFF    LEA ECX,DWORD PTR SS:[EBP-100]
0037B583  51               PUSH ECX
0037B584  FF55 D0          CALL DWORD PTR SS:[EBP-30]      ADVAPI32.CryptAcquireContextW
```

```
0037B596  52               PUSH EDX
0037B597  6A 00            PUSH 0
0037B599  6A 00            PUSH 0
0037B59B  68 03800000      PUSH 8003
0037B5A0  8B85 00FFFFFF    MOV EAX,DWORD PTR SS:[EBP-100]
0037B5A6  50               PUSH EAX
0037B5A7  FF55 D4          CALL DWORD PTR SS:[EBP-2C]      ADVAPI32.CryptCreateHash
```

```
0037B5B3  6A 01            PUSH 1
0037B5B5  8B4D 14          MOV ECX,DWORD PTR SS:[EBP+14]
0037B5B8  51               PUSH ECX
0037B5B9  8B55 10          MOV EDX,DWORD PTR SS:[EBP+10]
0037B5BC  52               PUSH EDX
0037B5BD  8B85 0CFFFFFF    MOV EAX,DWORD PTR SS:[EBP-F4]
0037B5C3  50               PUSH EAX
0037B5C4  FF55 D8          CALL DWORD PTR SS:[EBP-28]      ADVAPI32.CryptHashData
```

```
0037B5D6  51               PUSH ECX
0037B5D7  6A 01            PUSH 1
0037B5D9  8B95 0CFFFFFF    MOV EDX,DWORD PTR SS:[EBP-F4]
0037B5DF  52               PUSH EDX
0037B5E0  68 10660000      PUSH 6610
0037B5E5  8B85 00FFFFFF    MOV EAX,DWORD PTR SS:[EBP-100]
0037B5EB  50               PUSH EAX
0037B5EC  FF55 DC          CALL DWORD PTR SS:[EBP-24]      ADVAPI32.CryptDeriveKey
```

```
0037B5F5  8B8D 0CFFFFFF    MOV ECX,DWORD PTR SS:[EBP-F4]
0037B5FB  51               PUSH ECX
0037B5FC  FF55 E0          CALL DWORD PTR SS:[EBP-20]      ADVAPI32.CryptDestroyHash
```

现在发生了神奇的事情。加密密匙是之前提取的主文件的文件名（“3FKGytBDrBQsD6lyO3ahoqj.2tiUAHhzBB4k0X”）。

```
0037B60C  50              PUSH EAX
0037B60D  6A 00           PUSH 0
0037B60F  6A 01           PUSH 1
0037B611  6A 00           PUSH 0
0037B613  8B8D 04FFFFFF   MOV ECX,DWORD PTR SS:[EBP-FC]
0037B619  51              PUSH ECX
0037B61A  FF55 E4         CALL DWORD PTR SS:[EBP-1C]      ADVAPI32.CryptDecrypt
```

出现了一个无效乱码 PE。

```
Address   Hex dump                                        ASCII
02E90000  81 BB 00 4D 5A 90 00 03  00 00 00 82 04 00 30 FF  ¤.MZ.└...,┴.0ÿ
02E90010  FF 00 00 B8 00 38 2D 01  00 40 04 38 19 00 F0 00  ÿ...8- .@┴8↓.ð.
02E90020  0C 0E 1F 00 BA 0E 00 B4  09 CD 21 B8 00 01 4C CD  .♫.º♫.´.Í!¸. LÍ
02E90030  21 54 68 69 73 00 20 70  72 6F 67 72 61 6D 00 20  !This. program.
02E90040  63 61 6E 6E 6F 74 20 00  62 65 20 72 75 6E 20 69  cannot .be run i
02E90050  00 6E 20 44 4F 53 20 6D  6F 80 64 65 2E 0D 0D 0A  .n DOS mo€de....
02E90060  24 05 CE 80 FA CF 03 04  9B A1 50 05 03 20 01 97  $↓Î€úÏ♥♦›¡P↓ ♥ ù
```

由于缓冲区已被压缩，所以需要解压。这也是为什么需要分配新的存储区（未压缩缓冲区的大小也从文件中提取出来了）并调用 RtlDecompressBuffer 的原因。

0x102 是一种压缩格式，它是 0x100 | 0x2 0x100 的首字母缩写。0x100 是最高的压缩级别，而 0x02 是 LZNT1 压缩算法。

```
003796A0  8D4D FC         LEA ECX,DWORD PTR SS:[EBP-4]
003796A3  51              PUSH ECX
003796A4  8B55 CC         MOV EDX,DWORD PTR SS:[EBP-34]
003796A7  8B02            MOV EAX,DWORD PTR DS:[EDX]
003796A9  50              PUSH EAX
003796AA  8B4D C8         MOV ECX,DWORD PTR SS:[EBP-38]
003796AD  51              PUSH ECX                        Compressed Buffer
003796AE  8B55 CC         MOV EDX,DWORD PTR SS:[EBP-34]
003796B1  8B42 04         MOV EAX,DWORD PTR DS:[EDX+4]
003796B4  50              PUSH EAX
003796B5  8B4D D8         MOV ECX,DWORD PTR SS:[EBP-28]
003796B8  51              PUSH ECX                        Will contain uncompressed buffer
003796B9  68 02010000     PUSH 102                        Compression Format
003796BE  FF95 E8000000   CALL DWORD PTR SS:[EBP+E8]      RtlDecompressBuffer
```

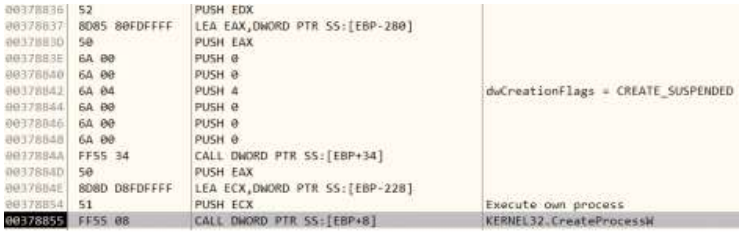
出现了混乱的有效 PE!

```
Address   Hex dump                                        ASCII
02EB0000  4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00  MZ.└...♦...ÿÿ..
02EB0010  B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ........@.......
02EB0020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
02EB0030  00 00 00 00 00 00 00 00  00 00 00 00 F0 00 00 00  ............ð...
02EB0040  0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  ♫.º♫.´.Í!¸.LÍ!Th
02EB0050  69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is program canno
02EB0060  74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run in DOS
```

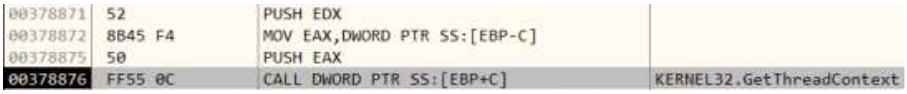
显然，该文件是已知的 Locky 变种 (SHA256: 31af9ea19741da26235b9f6e253da5112d27260545cf3034bd12ff36a8b65dad)

既然 shellcode 获得了所需的条件，接下来怎么操作呢？

shellcode 执行在 SUSPENDED_STATE 中运行时（例如，如果代码运行 calc.exe，它就再次执行 calc.exe）同样的程序。强烈指示挖空进程。



下一步是进入主线程环境，引导随后的执行活动。



——替代暂停进程内容

有趣的是，该程序不是典型的进程挖空。

要替换暂停进程的内容，代码需要再次映射 ntdll.dll 并使用搜索机制（CRC32）搜寻与 Win32API “ReadProcessMemory”等同的“NtReadVirtualMemory”系统调用号，然后用它来确定目的映像的基地址。

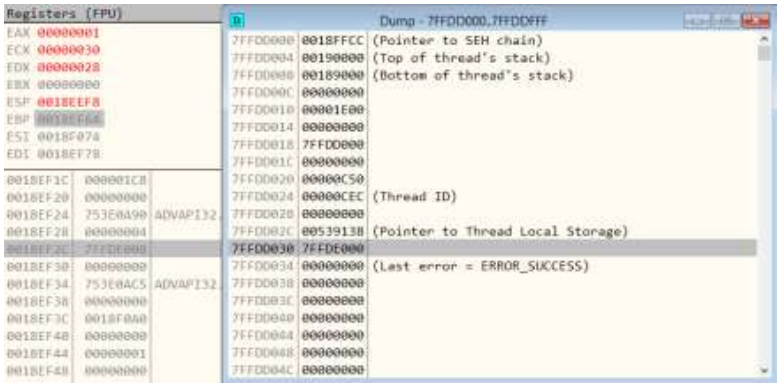
获得基地址后，代码再次搜索，但这次搜索的是“NtCreateSection”的系统调用号，然后利用 ACCESS_MASK of SECTION_MAP_WRITE | SECTION_MAP_READ | SECTION_MAP_EXECUTE 创建一个代码区段。

代码创建区段后，开始搜索“NtMapViewOfSection”的系统调用号以便把区段映射到之前创建的暂停进程内。（我在进程句柄传递到 NtMapViewOfSection 才知道这个情况）。

重复这一活动，但这次是映射区段到自己的地址空间内（因为该进程句柄是-1）。这意味着该区段在进程间共享。

下一步是映射解密 PE（Locky 早前解密的映像）到该区段，与映射到 ntdll.dll 的做法完全一致——解析 Locky 映像并复制数据头和区段。

现在，该区段在两个进程间共享并包含 Locky 的 PE。然后搜索与 Win32API “WriteProcessMemory” 等同的系统调用号 “NtWriteVirtualMemory”。然后利用该函数，代码用 LockyPE 映像库替换暂停进程中的映像库地址。



从上图，我们可以看到暂停进程的线程信息块（右边的窗口）和 30h 点到暂停程序的 PEB（进程环境块）。通过 NtWriteVirtualMemory 的参数（等同于 “WriteProcessMemory”）是 PEB+8h 的地址（+8h 是到 PEB 映像基地址的）。

一切就位后，然后使用 SetThreadContext 改变 CONTEXT 内部结构（CONTEXT 是之前用 GetThreadContext 提取的）的 EAX 了。

```
00368BE4  898D 40FBFFFF  MOV DWORD PTR SS:[EBP-4C0],ECX   Update CONTEXT.EAX = EntryPoint
00368BEA  8D95 90FAFFFF  LEA EDX,DWORD PTR SS:[EBP-570]
00368BF0  52             PUSH EDX
00368BF1  8B45 F4        MOV EAX,DWORD PTR SS:[EBP-C]
00368BF4  50             PUSH EAX
00368BF5  FF55 10        CALL DWORD PTR SS:[EBP+10]       KERNEL32.SetThreadContext
```

[EBP-4C0]现在包含 Locky 映像的入口点

```
0018F198  001C570D
```

AddressOfEntryPoint	0x0000570D (section:.text)

请注意，代码映射区段到暂停进程时，抵消物 0x1C0000 从 NtMapViewOfSection 返回。因此，0x1C0000 基本上是暂停进程内部的共享区段抵消物。

最后一步：利用上面提到的搜索机制照样搜寻“NtResumeThread”系统调用号。

```
02CE9EBD  68 C8CCE116   PUSH 16E1CCC8          CRC32 of "NtResumeThread"
02CE9EC2  E8 E0F8FFFF   CALL 02CE97A7          Map NTDLL and search for CRC32
```

代码为恢复线程获得系统调用号后，只需利用 SYSCALL 和 Locky 文件，然后在加密计算机整个文件系统之时开始执行调用。

——Cerber 和 Locky 的相似点

检查 Cerber 时（SHA256：dbd21df0f96f870875e4abafaa33b595df9e4da8b39ccc6c717fea9afcaedef3），我们注意到了两个版本含有相似的行为。

首先，最近的 Cerber 变种也来自于 NSIS 安装程序，有着同样的文件结构。

脚本（[NSIS].nsi）相同：

执行的代码仔细研究后，发现看起来完全一样，它们运行的 XOR 程序也相同，搜寻 DE-XOR 区段的字节标记。

```
00697660  3855 00     CMP BYTE PTR SS:[EBP],DL    CHECK FOR 0xED
00697663  74 10       JE SHORT 00697675
```

与之前的例子不同（0x4E），这次是 0xED。

XOR 进程也有相同的 XOR ESI，利用 EAX 查找 4 个密匙。

```
006976BF  33C6          XOR EAX,ESI
006976C1  3B45 04       CMP EAX,DWORD PTR SS:[EBP+4]
006976C4  79 05         JNS SHORT 006976CB
```

代码完成 XOR 所有事情后，我们可以看到它利用相同的 CRC32 检索机制搜寻系统调用号。

代码检索到系统调用号后，也需使用 Heaven’ s Gate 技术。

```
003D9E6D  6A 33           PUSH 33                    push 33h as segment
003D9E6F  E8 00000000     CALL 003D9E74
003D9E74  830424 05       ADD DWORD PTR SS:[ESP],5   set offset of 64-bit code
003D9E78  CB              RETF                       Far return
```

```
003D9EAF  8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]   move system call number to EAX
003D9EB2  0F05            SYSCALL
```

搜寻挖空进程的技术也与上面一样：

——在暂停状态下创建自己的映像作为新的进程

——利用 GetThreadContext 获取 CONTEXT

——创建一个区段

——映射区段到两个暂停进程 ,并利用 NtMapViewOfSection 系统调用号映射自己的进程（因此，区段在三个进程间共享）。

——解密 Cerber PE，然后映射到区段上（加密密匙也是它的文件名-“zXoueT8m.cdhNtye”）。

——准备 PEB 暂停进程内的 ImageBaseAddress 到 Cerber 上的 ImageBase。

——把 CONTEXT.EAX（包含加密点）改为 Cerber 的入口点

——恢复线程

NSIS 提取器

我们编写了一个工具 ,它利用 C 语言和 Python 从给定的 NSIS 安装程序中提取真正的勒索软件二进制文件。我们的 GitHub 上有它的源代码。

利用深度学习结合强大的研究能力，使得 Deep Instinct 能够提供新恶意软件变种无以匹敌的检测能力，给终端和移动设备提供强大的保护能力。

感染信标

Locky OSIRIS

c976db2208c3fda077da5cd51355f958417b7d3d180a817aaeb7a62acf9faf83

da1469e08123a829e5d33d0e51632953c3e0b36abec90cfe8ff5cb812f9d56e9

Cerber 5.0.1

993ee9f39003c5221f270846c0df668b4b3258e6f72ad6cc3c1f3e14c5f16ae9

dbd21df0f96f870875e4abafaa33b595df9e4da8b39ccc6c717fea9afcaedef3

Cerber 4.1.1

29744b8bab2f176444a8d614bfb96de05803585be50ebc9fa62bc2a027db96a3

Other versions of Cerber:

c27db0c832d5821454d1881d323a6745e8356fd531e7565809bc6cd99af6d682

0635dedd5b1e4b21e1324828608973926417e3b53900a7a8dc8ef8f0c068df2b

45ee98554aa6ea466a17609740f6e2f8dc4de11ebce3c3eb72d73c4fcbb16d1e

39a1eeda1f5c252a2daaa60609bb151bcdbe35ec753d7daad1904152509cf49

Dumped original version of Locky:

31af9ea19741da26235b9f6e253da5112d27260545cf3034bd12ff36a8b65dad