某大型炼钢厂内网蓝屏重启的应急响应之旅

作者:超弦 AI 攻防实验室

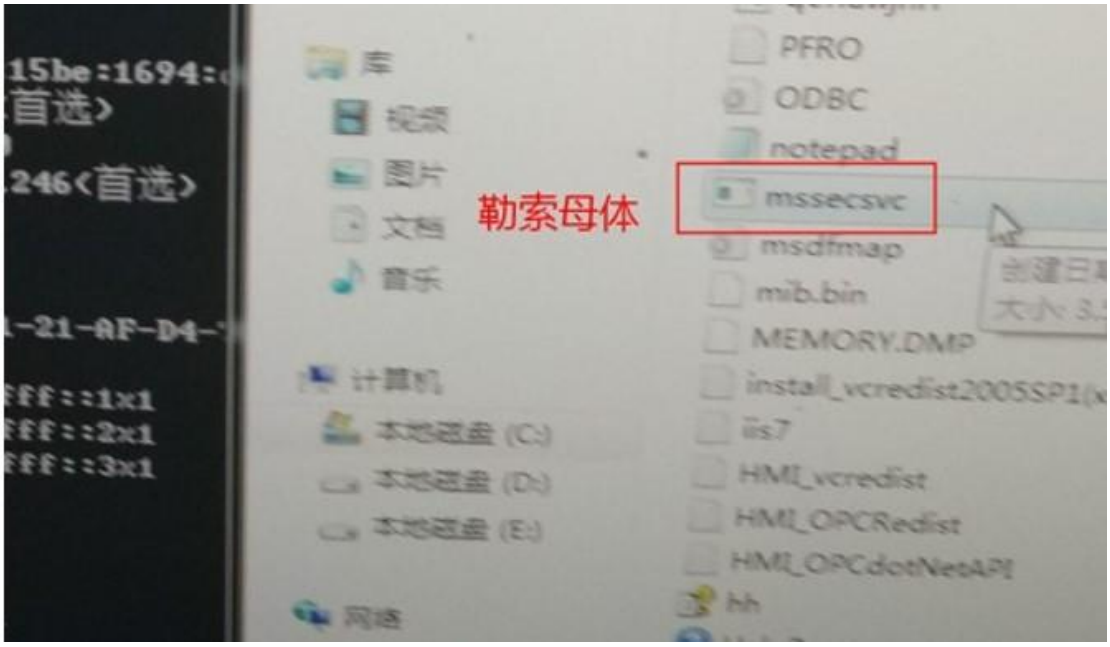
0x0 背景

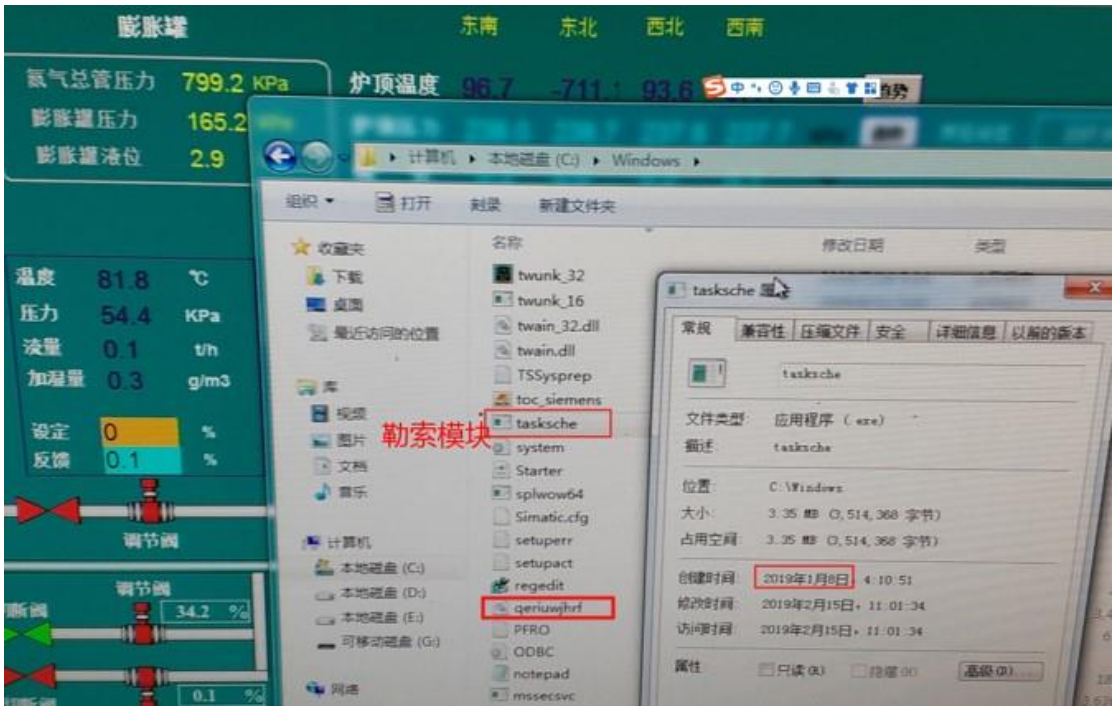
近日,我们实验室接到国内某大型炼钢厂的应急求救,据其描述其内网高炉车间主机于 2019 年 1 月 8 日开始出现蓝屏和一分钟倒计时重启两种现象,烧炉车间主机于 2 月 14 日也开始出现蓝屏和倒计时重启,在发生现象以后其有自行安装过杀毒软件进行扫描杀毒,但是问题还是时有发生,于是通过渠道联系到我们,希望得到彻底的定位与解决。

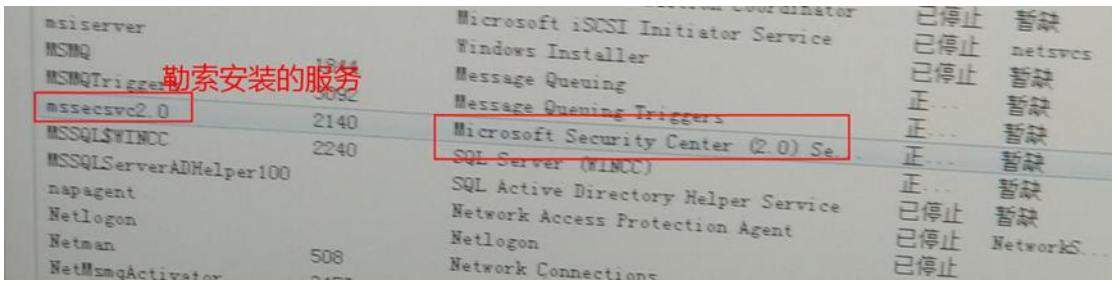
在接到通知后,粗略了解事件原委之后,我们内部应急响应小组立即就现场支持召开会议进行讨论,然后确定由我和另一名成员前往进行现场支持。根据我们之前的现场应急经验,我们怀疑这有可能是内网中了勒索蠕虫,蠕虫发动内网攻击导致,我们随即准备了内部勒索专杀工具以及应急响应专用 U 盘(含一键取证脚本)连夜赶往该炼钢厂。

0x1 取证与定位分析

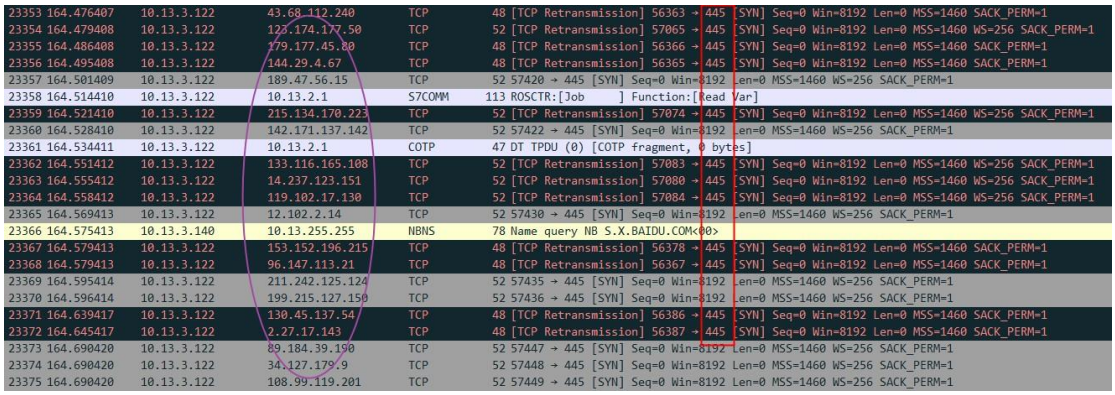
第二日早晨我们来到炼钢厂高炉中控室,首先与负责人进行交流,再次确认现象,然后在他们的允许下对发生问题的几台备用的机器进行信息取证(在不影响生产的前提进行应急响应),基于异常现象为导向,我们主要针对主机和网络两个层面进行取证,主机层面主要从 windows 日志、系统信息、进程、服务、蓝屏 dump 等关键地方入手;网络层面主要抓取内网流量包,通过主机信息分析,我们发现当前主机中一可疑进程 mssecsvc.exe 和可疑服务 mssecsvc2.0,根据我们的分析经验以及威胁情报,我们很快断定该主机感染了 Wannacry 勒索蠕虫(mssecsvc.exe 是 Wannacry 勒索母体),在系统目录下我们也发现该勒索母体、勒索模块 tasksche.exe 以及勒索模块的副本 qeriuwjhrf:



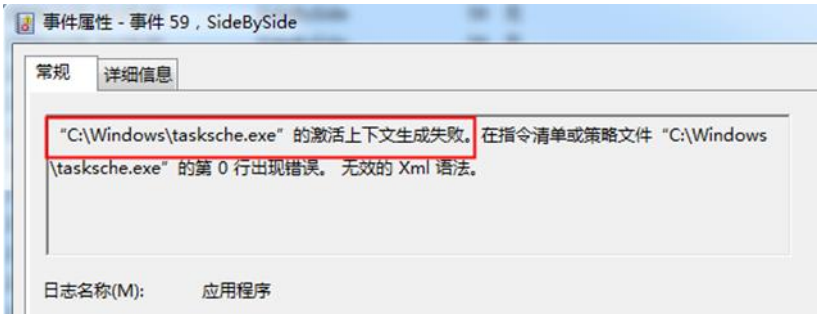




通过流量包能看出此感染主机正不断对内网和外网随机 IP 发送漏洞利用攻击包：



那么问题来了,既然已经感染了 Wannacry 为什么磁盘文件没有被加密?
继续分析,我们在日志中发现勒索模块 tasksche.exe 启动失败的信息:



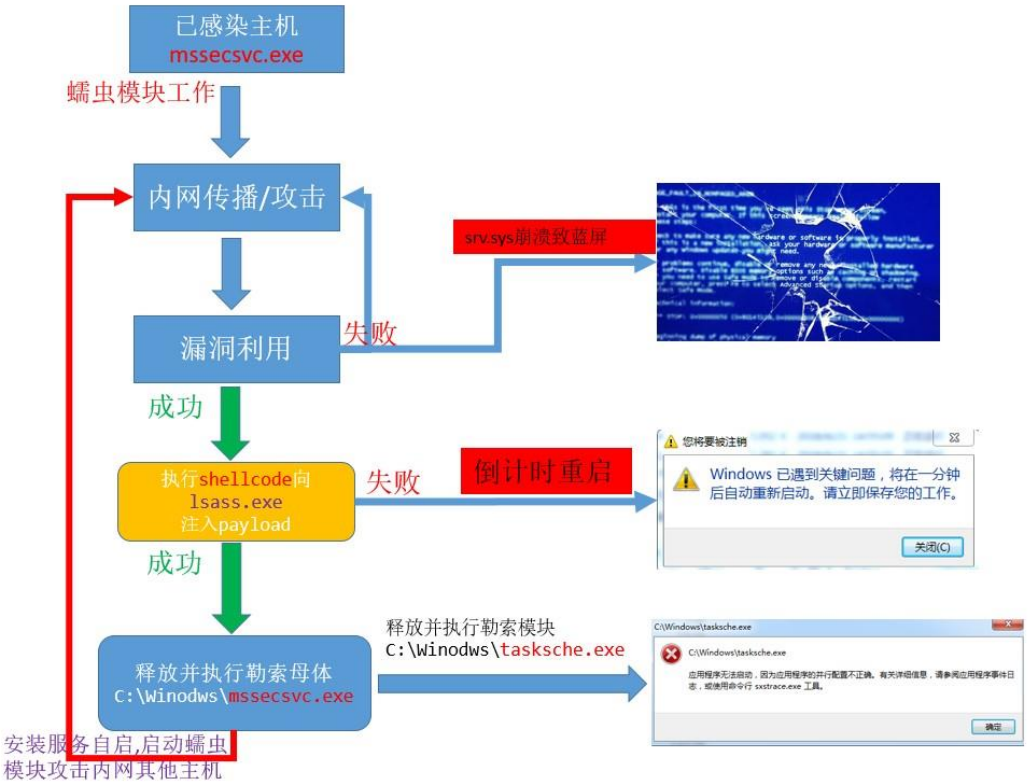
经过分析我们发现,与之前的 Wannacry 样本对比,该样本是被修改过的,其资源节中的勒索程序打包存在错误,导致勒索程序不能正常有效运行,因此炼钢厂“幸运”地逃过真正的勒索,避免了巨大的损失:

我们都知道 Wannacry 勒索蠕虫利用的是“永恒之蓝”MS17-010 漏洞,其会通过 445 端口向目标机器 SMB 服务器发送漏洞攻击包,对 srv.sy 驱动进行堆溢出,一旦利用失败便会导致驱动崩溃,造成蓝屏,通过蓝屏 dump 我们也确认了这一点:

```
FAULTING_IP:
srv!SrvOs2FeaToNt+19
b64942cc 668b4702        mov     ax,word ptr [edi+2]

MM_INTERNAL_CODE:  0

CUSTOMER_CRASH_COUNT:  1

DEFAULT_BUCKET_ID:  VISTA_DRIVER_FAULT

BUGCHECK_STR:  0x50

PROCESS_NAME:  System

CURRENT_IRQL:  0

LAST_CONTROL_TRANSFER:  from b649468b to b64942cc

STACK_TEXT:
b6c33b5c b649468b 87de75d0 cc0b8ffd 87792008 srv!SrvOs2FeaToNt+0x19
b6c33b7c b64ad771 87de75d0 b6c33bbc b6c33ba8 srv!SrvOs2FeaListToNt+0x9e
b6c33bb4 b64b533b 87dd4008 84275180 cc0a8008 srv!SrvSmbOpen2+0x93
b6c33bc8 b64b63aa b64836ec 87792008 b6485000 srv!ExecuteTransaction+0x101
b6c33c00 b647e530 87792008 8ae17058 8ae17020 srv!SrvSmbTransactionSecondary+0x2c5
b6c33c28 b648e9a0 00000000 8ae1cd48 00000000 srv!SrvProcessSmb+0x187
b6c33c50 84446013 00e17020 9fd58701 00000000 srv!WorkerThread+0x15c
b6c33c90 842eccd9 b648e844 8ae17020 00000000 nt!PspSystemThreadStartup+0x9e
00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x19


STACK_COMMAND:  kb

FOLLOWUP_IP:
srv!SrvOs2FeaToNt+19
b64942cc 668b4702        mov     ax,word ptr [edi+2]

SYMBOL_STACK_INDEX:  0

SYMBOL_NAME:  srv!SrvOs2FeaToNt+19

FOLLOWUP_NAME:  MachineOwner

MODULE_NAME: srv

IMAGE_NAME:  srv.sys

DEBUG_FLR_IMAGE_TIMESTAMP:  4dba2686

FAILURE_BUCKET_ID:  0x50_srv!SrvOs2FeaToNt+19

BUCKET_ID:  0x50_srv!SrvOs2FeaToNt+19

Followup: MachineOwner
---------
```

而对于倒计时重启主要是因为当漏洞利用成功后,将执行 shellcode 并通过 APC 将攻击载荷注入到系统关键进程 lsass.exe,企图实现无文件攻击，但一旦注入失败,便会导致 lsass.exe 进程崩溃,弹出一分钟倒计时重启的提示框。

为了便于读者理解,我们绘制了如下流程图:



如上图所示，在整个攻击过程中，只有漏洞利用和进程注入均执行成功,才不会引起蓝屏或重启现象，进而释放勒索模块 tasksche.exe,而该模块又是“无效的”,所以就不会表现出任何明显的现象,这也解释了为什么内网有的机器出现过现象、有的机器没有出现过现象，没有出现过现象不代表没有问题，恰恰是 Wannacry 攻击成功了,这个从日志、系统进程与服务以及相关特征文件上可以得到确认。

随后我们对发生过蓝屏和倒计时重启的主机所在的两个车间(高炉和烧炉)进行了全面排查与取证，在排查的过程中我们通过实例向负责人解释了“同样的系统配置为什么有的机器从没有发生过蓝屏重启”的疑问，最后我们确认了高炉车间网络于 2019 年 1 月 8 日凌晨 3 点多开始感染该勒索病毒,烧炉车间网络于 2019 年 2 月 7 日下午 3 点多开始感染,二者网段不通，由于 windows 日志记录时长较短，加之内网没有部署任何审计或防护软件，对于溯源到初始入侵向量比较困难,据负责人介绍整个炼钢厂内网与外网完全隔绝,我们推测可能是有操作人员有意或无意通过 u 盘等可移动存储器进行拷贝文件安装运行导致染毒，感染内网。

另外在全面排查与取证的过程中，我们发现客户有几台主机部署了某国内安全厂商的工业控制安全软件,在询问客户关于该工业安全软件的时候，客户问起:“你看，我们这几台机器部署了他们的工业安全软件，为什么还发生蓝屏与重启呀，是不是没防住?” 我们笑着告诉客户:“哈哈，的确是没防住，不是这不是它无能，而是漏洞利用超出了他防御的范围，该安全软件主要基于进程白名单的思想进行防御，防止白名单外的进程执行，区别于传统杀软的黑名单思想,然而该病毒在漏洞利用攻击过程中是没有进程产生的，其通过执行内存 shellcode 进行无文件攻击,所以其实漏洞利用和进程注入 2 个动作对它来说都是透明的，但是到后面释放勒索模块并执行的时候它就起作用了，你看，它的日志里这不是

记录着进程拦截信息嘛” 客户听了,挠了挠头略带疑问的说:”那到底该怎么在内网防范这种病毒攻击呀”，我们回答道:“这种勒索蠕虫的强大之处在于它的网络传播模块也就是蠕虫模块，其会利用漏洞迅速地在内网和外网中传播，并且漏洞利用攻击包一般杀软难以检测到，需要部署专业网络层的防护，比如防火墙/审计、IDS/IPS 等，然后再结合主机层的工业安全软件等进行组合拳式的立体防护。” 客户听后,点头表示明白了。随后客户又提起一事:“对了,我们在试用他们这工业安全软件的时候发现也有出现过蓝屏，是在系统启动的过程蓝屏的,您看看是否有时间也帮我们分析一下?” 我们连忙回道:”好，咱们先解决问题，这个回头我们帮你分析一下哈”; 客户露出了满意的表情☺

综上,经过我们的全面排查与取证分析，我们得出这样的结论:高炉车间和烧炉车间网络均已在不同时间点感染该变种 Wannacry 勒索蠕虫，并且在内网不断对外发送漏洞攻击包进行内网感染，现象主要是蓝屏和倒计时重启。

在取证与定位分析结束以后,我们向对方负责人进行汇报,经讨论一致,准备进入下一阶段:清除与恢复。

0x2 清除与恢复

在该阶段,我们主要是针对内网中的可操作的主机一台一台地进行“**断网隔离+手工杀毒+补丁免疫+重启恢复**”,原则是不影响生产且不安装额外软件等,很快便对车间内所有可操作的主机完成了病毒清除与生产恢复,随后我们给了客户一个内部病毒自动查杀与免疫工具,交待客户在其空闲的时候对其他没有处理的主机进行清除与恢复并反馈给我们结果,客户反馈结果良好。

0x3 蓝屏风波又起？某工业安全卫士惹的祸！

就在前几天炼钢厂负责人又联系我们说,我们处理过的某台机器在系统重启后登陆桌面的过程中又发生了蓝屏,而且这台机器装有某厂工业安全软件， 跟客户要来蓝屏 dump 后我们立即联合之前的 dump 进行综合分析,通过对之前客户反馈的启动过程中发生的蓝屏 dump 进行分析,我们发现其确实是由机器上安装的某工业安全卫士的驱动 AntiTP.sys 引发:

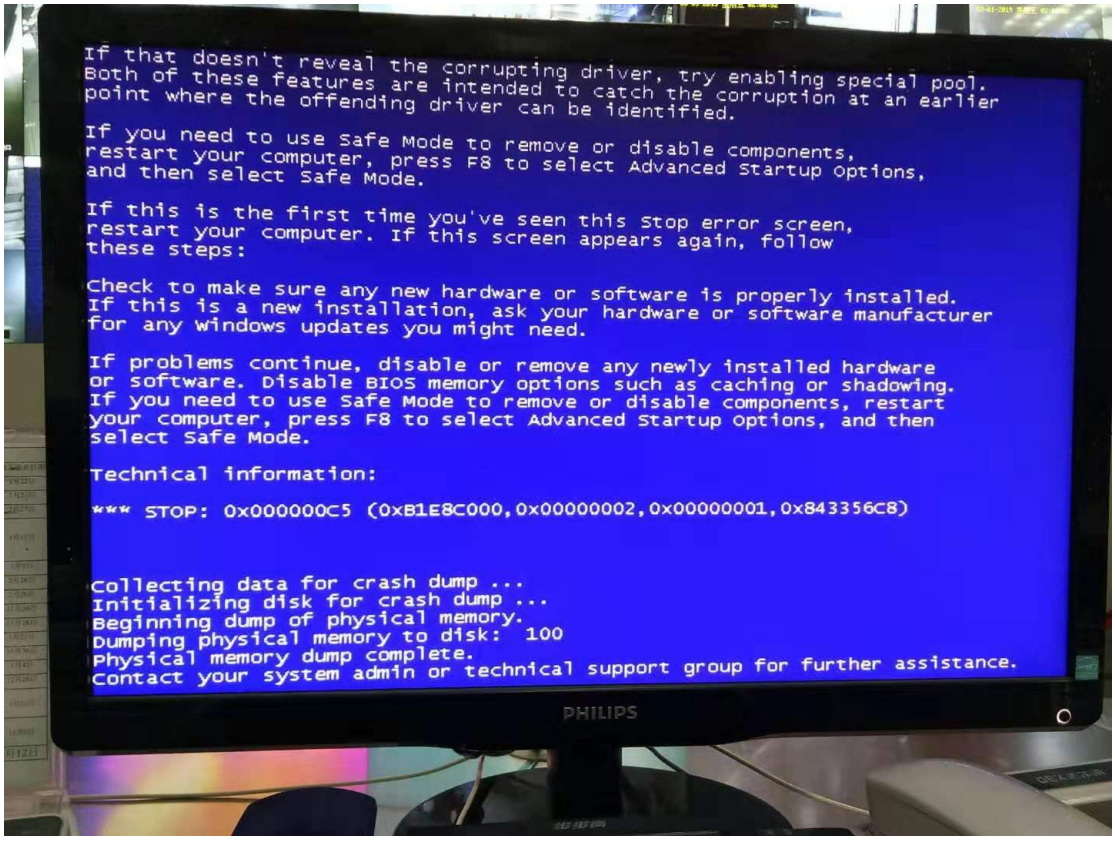
32 位机器:

```
CUSTOMER_CRASH_COUNT:  1

DEFAULT_BUCKET_ID:  VISTA_DRIVER_FAULT

BUGCHECK_STR:  0x8E

PROCESS_NAME:  svchost.exe

CURRENT_IRQL:  2

LAST_CONTROL_TRANSFER:  from 8436d243 to 8428d3f5

STACK_TEXT:
987d1690 8436d243 987d1760 987d1758 00000000 nt!ExpInterlockedPopEntrySListFault
987d16e0 844b4075 00000001 0000000b 67727453 nt!ExAllocatePoolWithTag+0x239
987d16f4 844b4330 0000000b b163dda2 844b42db nt!ExpAllocateStringRoutine+0x14
987d1734 95131a6d 987d1758 987d1760 00000001 nt!RtlUnicodeStringToAnsiString+0x55
WARNING: Stack unwind information not available. Following frames may be wrong.
987d1768 951370cb 882fe908 987d1fe8 00000001 AntiTP+0x1a6d
987d17a4 95135d07 987d1fe8 89cc46d8 874fb898 AntiTP+0x70cb
987d23fc 844b4ce3 00000494 00000204 00000001 AntiTP+0x5d07
987d24b4 844bcd22 874fb898 01cc46d8 987d2510 nt!PspInsertThread+0x5c7
987d24b8 874fb898 01cc46d8 987d2510 987d2580 nt!NtCreateUserProcess+0x751
987d24bc 01cc46d8 987d2510 987d2580 987d2420 0x874fb898
987d24c0 987d2510 987d2580 987d2420 987d24f8 0x1cc46d8
987d24c4 987d2580 987d2420 987d24f8 8438eae4 0x987d2510
987d2510 00000000 00090000 0008e000 00050000 0x987d2580


STACK_COMMAND:  kb

FOLLOWUP_IP:
AntiTP+1a6d
95131a6d ??              ???

SYMBOL_STACK_INDEX:  4

SYMBOL_NAME:  AntiTP+1a6d

FOLLOWUP_NAME:  MachineOwner

MODULE_NAME: AntiTP

IMAGE_NAME:  AntiTP.sys

DEBUG_FLR_IMAGE_TIMESTAMP:  5bd6cd10

FAILURE_BUCKET_ID:  0x8E_AntiTP+1a6d

BUCKET_ID:  0x8E_AntiTP+1a6d

Followup: MachineOwner
```

```
BUGCHECK_STR:  0xC5_2

CURRENT_IRQL:  2

FAULTING_IP:
nt!ExAllocatePoolWithTag+6bd
8432d6c8 832600          and     dword ptr [esi],0

CUSTOMER_CRASH_COUNT:  1

DEFAULT_BUCKET_ID:  VISTA_DRIVER_FAULT

PROCESS_NAME:  svchost.exe

TRAP_FRAME:  9782a624 -- (.trap 0xffffffff9782a624)
ErrCode = 00000002
eax=bc28e000 ebx=86e3a3c0 ecx=00005a2a edx=00000000 esi=bc28e000 edi=00001000
eip=8432d6c8 esp=9782a698 ebp=9782a6e0 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010286
nt!ExAllocatePoolWithTag+0x6bd:
8432d6c8 832600          and     dword ptr [esi],0    ds:0023:bc28e000=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 8432d6c8 to 8424cb5f

STACK_TEXT:
9782a624 8432d6c8 badb0d00 00000000 86e3b500 nt!KiTrap0E+0x1b3
9782a6e0 84474075 00000001 0000001c 67727453 nt!ExAllocatePoolWithTag+0x6bd
9782a6f4 84474330 0000001c be901b5d 844742db nt!ExpAllocateStringRoutine+0x14
9782a734 950f0a86 9782a750 9782a760 00000001 nt!RtlUnicodeStringToAnsiString+0x55
WARNING: Stack unwind information not available. Following frames may be wrong.
9782a768 950f60cb 883dad18 9782afe8 00000001 AntiTP+0x1a86
9782a7a4 950f4d07 9782afe8 8720ad40 87237030 AntiTP+0x70cb
9782b3fc 84474ce3 0000054c 000009b0 00000001 AntiTP+0x5d07
9782b4b4 8447cd22 87237030 0120ad40 9782b510 nt!PspInsertThread+0x5c7
9782b4b8 87237030 0120ad40 9782b510 9782b580 nt!NtCreateUserProcess+0x751
9782b4bc 0120ad40 9782b510 9782b580 9782b420 0x87237030
9782b4c0 9782b510 9782b580 9782b420 9782b4f8 0x120ad40
9782b4c4 9782b580 9782b420 9782b4f8 8434eb20 0x9782b510
9782b510 00000000 001d0000 001ce000 00190000 0x9782b580


STACK_COMMAND:  kb

FOLLOWUP_IP:
AntiTP+1a86
950f0a86 ??              ???

SYMBOL_STACK_INDEX:  4

SYMBOL_NAME:  AntiTP+1a86

FOLLOWUP_NAME:  MachineOwner

MODULE_NAME:  AntiTP

IMAGE_NAME:  AntiTP.sys

DEBUG_FLR_IMAGE_TIMESTAMP:  5bd6cd10

FAILURE_BUCKET_ID:  0xC5_2_AntiTP+1a86

BUCKET_ID:  0xC5_2_AntiTP+1a86
```

然后再对客户刚发过来的登陆桌面过程产生的蓝屏 dump 进行分析，发现栈回溯、错误代码等与之前的一致：

```
STACK_TEXT:
fffff880`05265c18 fffff800`042ecf69 : 00000000`0000000a fffff8a0`012d1000 00000000`00000002 00000000`00000001 : nt!KeBugCheckEx
fffff880`05265c20 fffff800`042ead88 : 00000000`00000001 fffff8a0`012d1000 00000000`00000200 fffffa80`0363e280 : nt!KiBugCheckDispatch+0x69
fffff880`05265d60 fffff800`04424e66 : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : nt!KiPageFault+0x448
fffff880`05265ef0 fffff800`04504bb1 : 00000000`00000001 fffff880`05266060 fffffa80`04c211da 00000000`00000000 : nt!ExAllocatePoolWithTag+0x326
fffff880`05265fe0 fffff880`063c664d : fffff880`05266080 fffff880`05266570 00000000`00000001 00000000`00000016 : nt!RtlUnicodeStringToAnsiString+0xc1
fffff880`05266040 fffff880`05266080 : fffff880`05266570 00000000`00000001 00000000`00000016 00000000`00420040 : AntiTP+0x764d
fffff880`05266048 fffff880`05266570 : 00000000`00000001 00000000`00000016 00000000`00420040 fffff880`05266570 : 0xfffff880`05266080
fffff880`05266050 00000000`00000001 : 00000000`00000016 00000000`00420040 fffff880`05266570 00000000`00170016 : 0xfffff880`05266570
fffff880`05266058 00000000`00000016 : 00000000`00420040 fffff880`05266570 00000000`00170016 fffff8a0`012cc510 : 0x1
fffff880`05266060 00000000`00420040 : fffff880`05266570 00000000`00170016 fffff8a0`012cc510 fffffa80`00210020 : 0x16
fffff880`05266068 fffff880`05266570 : 00000000`00170016 fffff8a0`012cc510 fffffa80`00210020 fffffa80`04b5c1a0 : 0x420040
fffff880`05266070 00000000`00170016 : fffff8a0`012cc510 fffffa80`00210020 fffffa80`04b5c1a0 00000000`00000000 : 0xfffff880`05266570
fffff880`05266078 fffff8a0`012cc510 : fffffa80`00210020 fffffa80`04b5c1a0 00000000`00000000 fffff880`063cc6a7 : 0x170016
fffff880`05266080 fffffa80`00210020 : fffffa80`04b5c1a0 00000000`00000000 fffff880`063cc6a7 00000000`00000000 : 0xfffff8a0`012cc510
fffff880`05266088 fffffa80`04b5c1a0 : 00000000`00000000 fffff880`063cc6a7 00000000`00000000 fffffa80`04c20dca : 0xfffffa80`00210020
fffff880`05266090 00000000`00000000 : fffff880`063cc6a7 00000000`00000000 fffffa80`04c20dca fffff880`05266e01 : 0xfffffa80`04b5c1a0
```



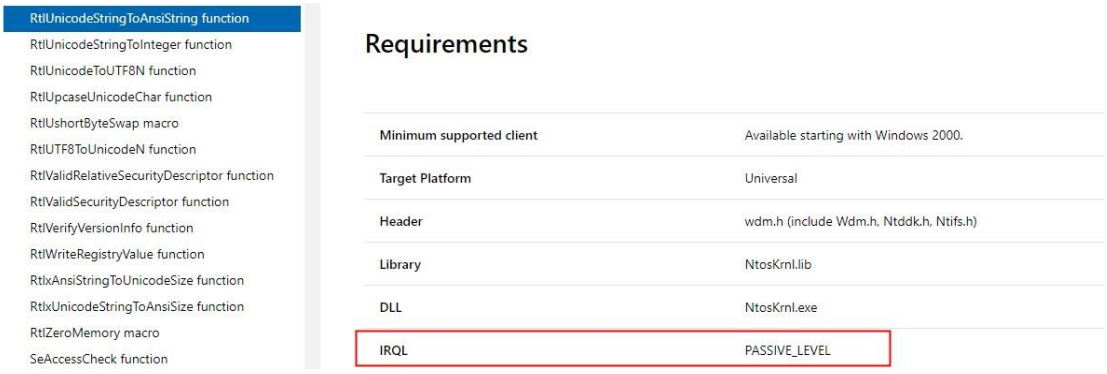
客户一开始怀疑是不是病毒没有处理干净或者又有新的病毒,我们赶紧回复:“没有,病毒已经清除了,这次蓝屏是由你们安装的那个工业安全卫士的一个驱动导致的，而且之前你提到的启动过程发生的蓝屏均是由它导致的!”，客户感叹道:“这软件有 bug 啊,得赶紧告诉他们开发加班修改！”

0x4　驱动蓝屏根源探索

为了搞清楚驱动为何频频导致蓝屏，我们决定一探究竟，Windbg 载入蓝屏 dump 如下:

```
DRIVER_CORRUPTED_EXPOOL (c5)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is
caused by drivers that have corrupted the system pool.  Run the driver
verifier against any new (or suspect) drivers, and if that doesn't turn up
the culprit, then use gflags to enable special pool.
Arguments:
Arg1: b1e8c000, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000001, value 0 = read operation, 1 = write operation
Arg4: 843356c8, address which referenced memory

Debugging Details:
------------------


BUGCHECK_STR:  0xC5_2

CURRENT_IRQL:  2

FAULTING_IP:
nt!ExAllocatePoolWithTag+6bd
843356c8 832600          and     dword ptr [esi],0

DEFAULT_BUCKET_ID:  VISTA_DRIVER_FAULT

PROCESS_NAME:  services.exe

TRAP_FRAME:  92a5d624 -- (.trap 0xffffffff92a5d624)
ErrCode = 00000002
eax=b1e8c000 ebx=86e38140 ecx=b1e8c002 edx=00000000 esi=b1e8c000 edi=00001000
eip=843356c8 esp=92a5d698 ebp=92a5d6e0 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010286
nt!ExAllocatePoolWithTag+0x6bd:
843356c8 832600          and     dword ptr [esi],0    ds:0023:b1e8c000=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 843356c8 to 8425306f

STACK_TEXT:
92a5d624 843356c8 badb0d00 00000000 86e39280 nt!KiTrap0E+0x1b3
92a5d6e0 8447d669 00000001 00000017 67727453 nt!ExAllocatePoolWithTag+0x6bd
92a5d6f4 8447d924 00000017 bbbf3339 8447d8cf nt!ExpAllocateStringRoutine+0x14
92a5d734 9459aa6d 92a5d758 92a5d760 00000001 nt!RtlUnicodeStringToAnsiString+0x55
WARNING: Stack unwind information not available. Following frames may be wrong.
92a5d768 945a00cb 881ca4f8 92a5dbd8 00000001 AntiTP+0x1a6d
92a5d7a4 9459ed1b 92a5dbd8 8a982d28 8786b030 AntiTP+0x70cb
92a5e3fc 8447e317 00000260 00000ae0 00000001 AntiTP+0x5d1b
92a5e4b4 84486e28 8786b030 01982d28 92a5e510 nt!PspInsertThread+0x5c6
92a5ebc0 8dd4b9d1 0012f258 0012f234 02000000 nt!NtCreateUserProcess+0x751
92a5ec00 8424fdb6 0012f258 0012f234 02000000 bd0001+0x79d1
92a5ec00 774a6c74 0012f258 0012f234 02000000 nt!KiSystemServicePostCall
0012f578 00000000 00000000 00000000 00000000 0x774a6c74


STACK_COMMAND:  kb

FOLLOWUP_IP:
AntiTP+1a6d
9459aa6d 85c0            test    eax,eax
```

可以看到栈回溯显示的调用路径跟之前的 dump 都是一致,都是在内核态字符串转换函数 RtlUnicodeStringToAnsiString 的内部调用路线中发生崩溃的,而且 BugCheck 分析提示我们当前函数的调用者所处的中断优先级(IRQL)太高,2 是 DISPATCH_LEVEL.查看 MSDN 中关于 RtlUnicodeStringToAnsiString 函数的说明我们可以看到微软指明该函数需要在 PASSIVE(0)中断级别下进行使用:



然后从栈回溯我们看到最后一次调用的是 nt!KiTrap0E,这是 windows 内核中的缺页异常处理函数,也就是说发生了缺页中断,为什么会发生缺页中断？因为驱动的想要访问的代码数据页不在真实内存(非分页内存)中,需要从虚拟内存(分页内存)中换出,故会触发缺页中断,然后我们看到堆栈回溯显示,RtlUnicodeStringToAnsiString 内部调用了内核态内存分配函数 ExAllocatePoolWithTag:

```
PVOID ExAllocatePoolWithTag(
  __drv_strictTypeMatch(__drv_typeExpr)POOL_TYPE PoolType,
  SIZE_T                                         NumberOfBytes,
  ULONG                                          Tag
);
```

而且其 PoolType 参数为 1 即分页内存 PagedPool:

```
ChildEBP RetAddr  Args to Child
92a5d624 843356c8 badb0d00 00000000 86e39280 nt!KiTrap0E+0x1b3
92a5d6e0 8447d669 00000001 00000017 67727453 nt!ExAllocatePoolWithTag+0x6bd
92a5d6f4 8447d924 00000017 bbbf3339 8447d8cf nt!ExpAllocateStringRoutine+0x14
92a5d734 9459aa6d 92a5d758 92a5d760 00000001 nt!RtlUnicodeStringToAnsiString+0x55
WARNING: Stack unwind information not available. Following frames may be wrong.
92a5d768 945a00cb 881ca4f8 92a5dbd8 00000001 AntiTP+0x1a6d
92a5d7a4 9459ed1b 92a5dbd8 8a982d28 8786b030 AntiTP+0x70cb
92a5e3fc 8447e317 00000260 00000ae0 00000001 AntiTP+0x5d1b
92a5e4b4 84486e28 8786b030 01982d28 92a5e510 nt!PspInsertThread+0x5c6
92a5ebc0 8dd4b9d1 0012f258 0012f234 02000000 nt!NtCreateUserProcess+0x751
92a5ec00 8424fdb6 0012f258 0012f234 02000000 bd0001+0x79d1
92a5ec00 774a6c74 0012f258 0012f234 02000000 nt!KiSystemServicePostCall
0012f578 00000000 00000000 00000000 00000000 0x774a6c74
```

enum POOL_TYPE

```
typedef enum _POOL_TYPE
{
        NonPagedPool = 0,
        PagedPool = 1,
        NonPagedPoolMustSucceed = 2,
        DontUseThisType = 3,
        NonPagedPoolCacheAligned = 4,
        PagedPoolCacheAligned = 5,
        NonPagedPoolCacheAlignedMustS = 6,
        MaxPoolType = 7,
        NonPagedPoolSession = 32,
        PagedPoolSession = 33,
        NonPagedPoolMustSucceedSession = 34,
        DontUseThisTypeSession = 35,
        NonPagedPoolCacheAlignedSession = 36,
        PagedPoolCacheAlignedSession = 37,
        NonPagedPoolCacheAlignedMustSSession = 38
} POOL_TYPE;
```

我们又发现 MSDN 中特别强调 ExAllocatePoolWithTag 的调用者如果处于 DISPATCH_LEVELA(2)的中断级别的时候只能使用非分页内存,只有中断级别小于 DISPATCH_LEVEL(或<=APC_LEVEL(1))的时候才可使用任意 Pool Type:

Callers of **ExAllocatePoolWithTag** must be executing at IRQL <= DISPATCH_LEVEL. A caller executing at DISPATCH_LEVEL must specify a **NonPaged***Xxx* value for *PoolType*. A caller executing at IRQL <= APC_LEVEL can specify any **POOL_TYPE** value, but the IRQL and environment must also be considered for determining the page type.

Allocating Memory with ExAllocatePoolWithTag

Drivers can also call ExAllocatePoolWithTag, specifying one of the following system-defined POOL_TYPE values for the *PoolType* parameter:

- *PoolType* = **NonPagedPool** for any objects or resources not stored in a device extension or controller extension that the driver might access while it is running at IRQL > APC_LEVEL.

  For this *PoolType* value, ExAllocatePoolWithTag allocates the amount of memory that is requested if the specified *NumberOfBytes* is less than or equal to PAGE_SIZE. Otherwise, any remainder bytes on the last-allocated page are wasted: inaccessible to the caller and unusable by other kernel-mode code.

  For example, on an x86, an allocation request of 5 kilobytes (KB) returns two 4-KB pages. The last 3 KB of the second page is unavailable to the caller or another caller. To avoid wasting nonpaged pool, the driver should allocate multiple pages efficiently. In this case, for example, the driver could make two allocations, one for PAGE_SIZE and the other for 1 KB, to allocate a total of 5 KB.

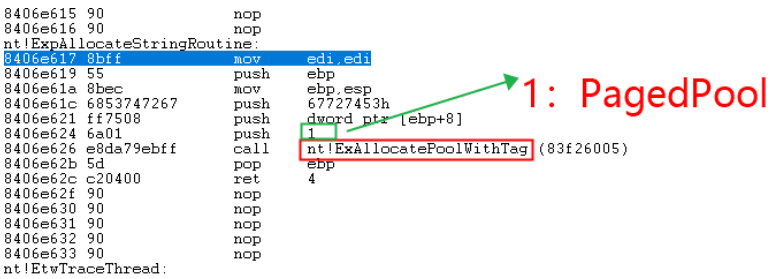
  Note Starting with Windows Vista, the system automatically adds the additional memory so two allocations are unnecessary.

- *PoolType* = **PagedPool** for memory that is always accessed at IRQL <= APC_LEVEL and is not in the file system's write path.

然而 dump 给我们的信息是驱动在调用 ExAllocatePoolWithTag 的时候处于 DISPATCH 中断级别,而且使用的是分页内存..这不是无视微软权威嘛..为了更一步确认驱动代码逻辑,我们开始通过 windbg + IDA 对 AntiTP.sys 进行逆向,通过 IDA 代码交叉引用很快能定位到字符串匹配函数 StringMatchW(),该函数主要是将当前进程和白名单中进程的 Unicode 路径字符串转换成 Ansi 字符串,然后再进行比较,来到 StringMathW 的上一层调用函数 FindExistInAllowProcessList(),我们发现其在进行字符串匹配函数 StringMatchW 调用之前,使用了 windows 自旋锁来进行内核同步(自旋锁是一种在内核定义,只能在内核态下使用的同步机制。自旋锁用来保护共享数据或者资源,使得并发执行的程序或者在高优先级 IRQL 的对称多处理器的程序能够正确访问这些数据。)

```
char __stdcall FindExistInAllowProcessList(void *FilePath)
{
  char *v1; // esi
  wchar_t *v3; // ebx
  unsigned int v4; // eax
  _UNICODE_STRING nowFile; // [esp+4h] [ebp-1Ch]
  _UNICODE_STRING ptfile; // [esp+Ch] [ebp-14h]
  PKSPIN_LOCK SpinLock; // [esp+14h] [ebp-Ch]
  unsigned int i; // [esp+18h] [ebp-8h]
  char ret; // [esp+1Fh] [ebp-1h]

  v1 = (char *)DeviceObject->DeviceExtension + 4;
  ret = 0;
  if ( !FilePath )
    return 0;
  SpinLock = (PKSPIN_LOCK)(v1 + 12);
  i = 0;
  v1[16] = KfAcquireSpinLock((PKSPIN_LOCK)v1 + 3);// increase the IRQL to DISPATCH_LEVEL(2) by using windows spin lock
  if ( *((_WORD *)v1 + 3) > 0u )
  {
    v3 = (wchar_t *)(v1 + 803860);
    while ( i < 0x100 )
    {
      if ( wcscmp(v3, (const unsigned __int16 *)byte_18630) )
      {
        RtlInitUnicodeString(&ptfile, v3);
        RtlInitUnicodeString(&nowFile, (PCWSTR)FilePath);
        if ( !wcsicmp(v3, (const wchar_t *)FilePath)
          || SpyFindSubString(&nowFile, &ptfile)
          || StringMatchW(v3, (unsigned __int16 *)FilePath, 1) )
        {
          ret = 1;
          break;
        }
      }
```

这与 dump 提供给我们的信息一致,发生问题的时候 IRQL 确实是 2,然后我们用 windbg 结合 vmware 进行 win7 内核的双机调试,加载 AntiTP.sys 然后在关键地方下断点来到 ExAllocatePoolWithTag 的函数内部,发现该函数默认分配的是分页内存☹:



而该驱动在调用 RtlUnicodeStringToAnsiString 之前又没有手动进行非分页内存的分配,且 IRQL 已经提高到 DISPATCH 级别,故容易导致蓝屏死机。

说到这里，读者可能会有如下 2 个问题，

问题 1:为什么分页内存在 DPC 级别下容易导致蓝屏?

因为驱动的字符串转换相关代码数据是存放在分页内存中的,当驱动需要调用相关函数或者访问该代码页时,如果该页内存数据不在物理内存中，也就是被交换到了虚拟内存页面文件中,将触发内存缺页中断，windows 将会试图访问虚拟内存页面文件 pagefile.sys，把被交换出的数据读入物理内存中,然而 windows 的缺页中断处理程序是运行在 DISPATCH_LEVEL 的级别的，我们的驱动此刻也是运行在 DISPATCH_LEVEL 级别,故这里我们的驱动不会被缺页中断打断进行内存页的读取，EIP 就会访问到一个错误的内存地址（很可能读/写到了其他进程,也可能是系统中重要的数据）上,用户层的程序内存访问违例的异常操作系统能捕捉到反馈给用户,而在驱动内核层内存访问错误,这是一个很严重的问题,操作系统发现立即就以蓝屏的方式处理了.

问题 2: 为什么不是必现,只是偶尔发生且主要在启动过程中?
其实不是没有问题,而是驱动的代码数据虽然是分页内存,但是没有被交换出去,还是在物理内存中，访问时不会引发缺页中断,也就没有表现出问题,但是，一旦某一刻当 windows 发现物理内存不够了，根据内存调度算法恰好把你的代码数据页交换到虚拟内存页面,这个时候基本就会引发蓝屏！而且启动加载过程中,驱动的相关代码数据处于虚拟内存中的概率较大(尚未完全加载)，这时候只要进行相关操作也就容易引起缺页中断致蓝屏!

0x5 驱动 bug 修改建议

1. 在调用字符串 RtlUnicodeStringToAnsiString 的时候降低 IRQL,使其<2
2. 不降低 IRQL,在调用字符串函数 RtlUnicodeStringToAnsiString 之前手工分配非分页内存!

这里我们推荐方法 2,比较符合安全编码的规范,驱动开发一定要谨慎!

0x6 事件总结

近几年工业安全事件频发， 由于受工业企业自身生产业务以及企业安全管理意识与缺乏的制约， 工业内网终端主机常常是处于“无补丁”、“无防护”、“裸奔”的脆弱状态，对于联网的工业内网来说，极其容易遭受黑客入侵，染毒感染大片工业厂区内网；对于与外网隔绝的工业内网来说，由于主机缺乏有效的终端安全防护措施(主机安全卫士等)， 加之对于 u 盘等可移动存储设备的使用缺乏安全管理与限制，极容易出现“内部员工到互联网不正规网站下载带毒的应用软件，然后通过 u 盘拷贝至内网终端主机上进行安装，进而感染整个内网”的现象。去年 8 月全球最大的半导体芯片厂商台积电爆发的内网 Wannacry 勒索感染事件，后来经证实属于内部员工安装新设备存有疏忽导致。

工业环境不同于传统的网络环境，发生问题的主机常常承载着生产业务，而且有的还没有备份机，因此针对工业安全事件的应急响应事件要格外谨慎，对案发现场把握“尽可能不影响生产” 、“绿色多维取证”、“简单快捷高效”、 “顺藤摸瓜”等原则进行事件响应， 以问题或现象为导向， 进行关联分析与溯源定位，复盘攻击路径与攻击画像。

针对目前国内工控环境存在的安全现状，我们实验室给工业企业如下的安全防护建议：

1.对员工进行安全教育，全面提高企业整体的安全意识
2.使用最小特权原则， 减少“内鬼”进行”恶意”操作的概率
3.部署工业内网终端主机安全软件和内网威胁管理系统等解决方案
4.调整内网不同厂区网络架构，部署工业防火墙等网络层防护方案
5.及时备份数据，做好容灾备份、及时更新系统与软件补丁，禁用高危端口等
6.与安全厂商进行合作，定期进行安全检查与攻防演练培训等

0x7 参考链接

1、
https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-exallocatepoolwithtag
2、
https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-rtlunicodestringtoansistring
3、https://blog.csdn.net/yw1621/article/details/64185513
4、https://www.xianjichina.com/news/details_66443.html
5、https://www.freebuf.com/news/139809.html
6、http://plc.gkzhan.com/news/4614.html
7、https://www.freebuf.com/articles/system/134578.html
8、http://www.antiy.com/response/wannacry.html

IoCs

mssecsvc.exe 0C694193CEAC8BFB016491FFB534EB7C
tasksche.exe 7F7CCAA16FB15EB1C7399D422F8363E8