SKREAM（二）:内存地址的随机分配

websni****@outlo / 2018-11-09 09:00:00 / 浏览数 2252 技术文章 技术文章 顶(0) 踩(0)

0x01 关于SKREAM

在前一篇文章中，我们讨论了内核池溢出漏洞，并提出了一种新的缓解方法，旨在防御Windows 7和8系统上使用特定溢出技术。该技术已应用到我们的SKREAM工具包里。

尽管我们在Windows
8.1中缓解了这种攻击手法(在0xbad0b0b0中构建恶意OBJECT_TYPE结构)，但是内存溢出漏洞仍屡禁不止，道高一尺，魔高一丈。利用的手法也再不断革新。因此我们也希

溢出成功有几个必要的前提。攻击者必须能够找到一个关键的地址来构建溢出缓冲区，并且准确地知道应该写入哪些数据，哪些需要保持其余数据不变。放错字节或是放错位

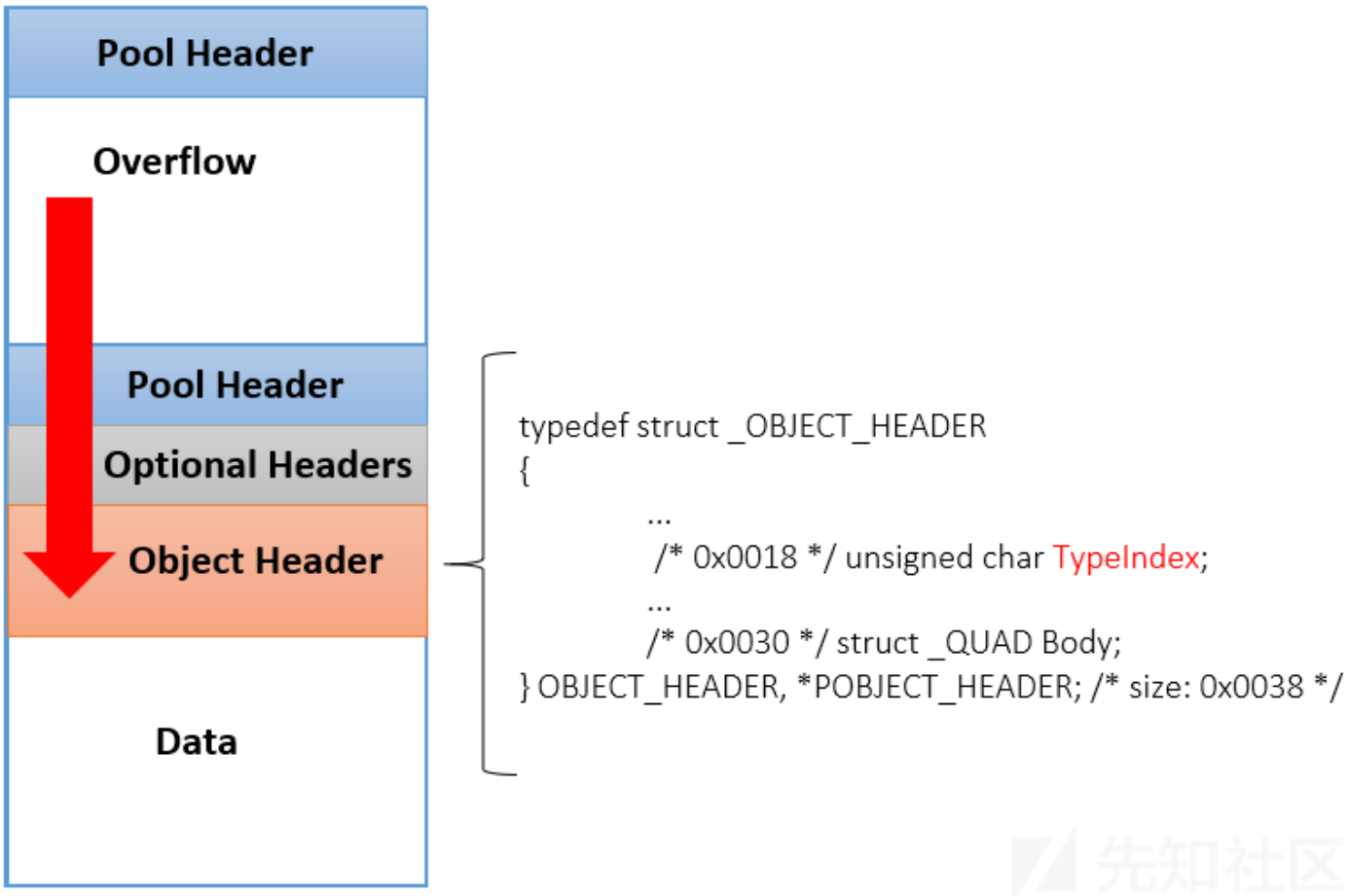

图1 内存溢出。比如在进行类型索引覆盖攻击时，该漏洞试图设置ObjectHeader。下一个池块的类型索引为0或1。
为了实现这个目标它必须计算ObjectHeader从溢出缓冲区开始的准确距离，以及TypeIndex的偏移量。

因为攻击需要精确到每一个字节，所以可以在池分配时引入随机分配来干扰这类漏洞。这里提供两种思路，一是选择转移(或隔离)分配，二是"膨胀"分配。两种思路的最终目

0x02 PoolSlider

下文中将以笔者个人对这项技术的理解"内存隔离"来叙述

如WDK文档里所说的一样，x64架构上的内存分配器所分配的字节长度必须四舍五入到16字节(x86架构上8字节)。这就意味着任何请求大小如果不足16个字节的整数倍的话

```
kd> bp nt!ExAllocatePoolWithTag ".if (@rdx % 0x10 == 0) { g; }"
kd> g
nt!ExAllocatePoolWithTag:
fffff800`029be0f0 fff5            push    rbp
kd> rrdx
rdx=0000000000000068
kd> pt
nt!ExAllocatePoolWithTag+0x21d:
fffff800`029be30d c3              ret
kd> !pool @rax
Pool page fffffa8001825b90 region is Nonpaged pool
 fffffa8001825000 size:   30 previous size:    0  (Allocated)  MmNo
 fffffa8001825030 size:  b50 previous size:   30  (Free)       ....
*fffffa8001825b80 size:   80 previous size:  b50  (Allocated) *MmDp
              Pooltag MmDp : Lost delayed write context, Binary : nt!mm
 fffffa8001825c00 size:   20 previous size:   80  (Free )  MmLd
 fffffa8001825c20 size:   80 previous size:   20  (Allocated)  MmIo
 fffffa8001825ca0 size:   80 previous size:   80  (Allocated)  MmIo
 fffffa8001825d20 size:   80 previous size:   80  (Allocated)  MmIo
 fffffa8001825da0 size:   80 previous size:   80  (Allocated)  MmIo
 fffffa8001825e20 size:   a0 previous size:   80  (Allocated)  Vadl
 fffffa8001825ec0 size:   90 previous size:   a0  (Allocated)  MmLp
 fffffa8001825f50 size:   30 previous size:   90  (Allocated)  MmNo
 fffffa8001825f80 size:   30 previous size:   30  (Allocated)  MmNo
 fffffa8001825fb0 size:   50 previous size:   30  (Allocated)  MmMl
```

图2 rdx大小为0x68字节的分配请求最后的实际大小是0x80字节:即请求头(0x10) +请求大小(0x68) +填充(0x8)

很明显如果想在这种条件下实现溢出，攻击者必须要考虑到字节不足而填充的部分。比如图2中尽管开发者只请求了0x68但是有0x70字节在到达下一个池分配之前必须被覆盖

在我们的内存池隔离保护技术中，我们可以利用了这个填充和"隔离"这两特性，让指针以随机数的形式返回给调用者。这样一来，只要我们混淆内存池的开头创建的填充字节

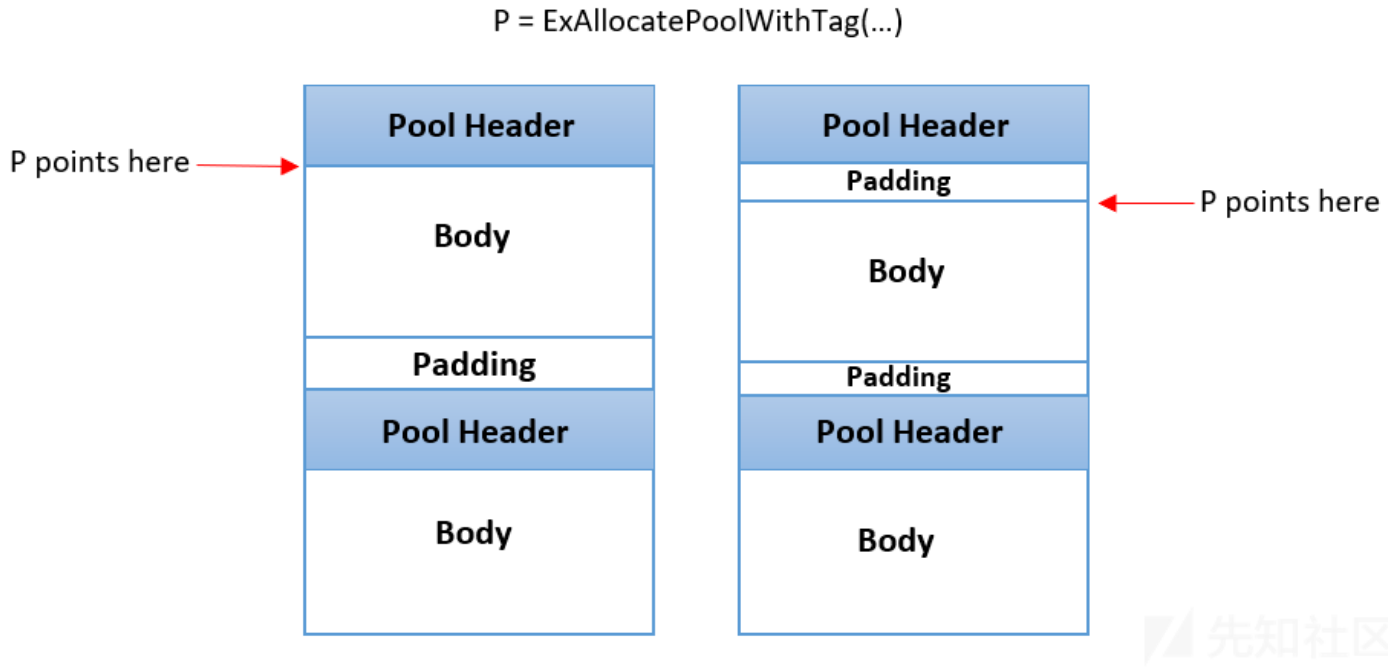

图3 有隔离(右)和无隔离(左)。

现在我们通过SKREAM扩展来监听图像加载事件[图像加载事件]，并在每个新加载驱动的`ExAllocatePoolWithTag`上放一个IAT钩子，从而实现了防御。每当内存池分配时

```
void ExAllocatePoolWithTag_Hook(
              POOL_TYPE PoolType,
              SIZE_T NumberOfBytes,
              ULONG Tag)
{
       PVOID P = ExAllocatePoolWithTag(PoolType, NumberOfBytes, Tag);
       ULONG Padding = POOL_GRANULARITY - (NumberOfBytes % POOL_GRANULARITY);
       ULONG r = rand(1, Padding);
       return (CHAR *)P + r;
}
```

0x03 处理释放

通过将返回的指针向前推进后，我们破坏了内存池的可预测性，这不仅仅是攻击者。假设内存池管理器返回给调用者的指针前面还有一个描述分配的POOL_HEADER结构。这 **- sizeof(POOL_HEADER)**搜索相关的池头数据。但是当使用内存池隔离技术时，假设完全没用，BAD_POOL_HEADER会导致系统崩溃。

为了正确地处理释放，我们必须在ExFreePoolWithTag上多放一个IAT钩子，并在处理释放前将指针重新对齐到16个字节。

```
void ExFreePoolWithTag_Hook(PVOID P, ULONG Tag)
{
       P = (PVOID)(ULONG_PTR)P & 0xfffffffffffffff0;  // align to 0x10
       ExFreePoolWithTag(P, Tag);
}
```

0x04 其他问题

在测试内存池隔离技术时，我们还遇到了一些问题。有些问题很容易搞定，而有些问题仍然对这种防御带来严重威胁:

- 使用ExAllocatePoolWithTag分配，ExFreePool释放，反之亦然。

  同时在ExAllocatePool和ExFreePool上放钩子，并在Ex{Allocate, Free}PoolWithTag处进行同样的随机/重新对齐处理。

- 分配带有ExAllocatePool(WithTag)的字符串并使用RtlFree{Ansi, Unicode}String释放。

  这种写法很烦躁，字符串应该交给对应的进程来分配。这些内部的释放函数将字符串对象的"缓冲区"元素转给ExFreePool(WithTag)，如果此时指针没有16个字节，就 Unicode}String上放个IAT钩子，使用在ExFreePool(WithTag)中一样的手法来重新对齐指针。

- 当一个驱动正在分配内存遇上另一个驱动释放内存。

  目前遇到的最复杂的情况是一个驱动分配内存时碰上另一个驱动(通常是NTOS)释放内存。在这种情况下，当释放驱动程序没有放钩子时，不能在调用ExFreePool之前重

```
1: kd> .bugcheck
Bugcheck code 000000C2
Arguments 00000000`00000007 00000000`0000109b 00000000`6c420303 fffff8a0`007d75d2
1: kd> k
 # Child-SP          RetAddr           Call Site
00 fffff880`02fe6ef8 fffff800`029bccc2 nt!RtlpBreakWithStatusInstruction
01 fffff880`02fe6f00 fffff800`029bdaae nt!KiBugCheckDebugBreak+0x12
02 fffff880`02fe6f60 fffff800`028cafc4 nt!KeBugCheck2+0x71e
03 fffff880`02fe7630 fffff800`029ffbf9 nt!KeBugCheckEx+0x104
04 fffff880`02fe7670 fffff800`02cba73c nt!ExFreePool+0xcb1
05 fffff880`02fe7720 fffff800`02cbc654 nt!PipProcessStartPhase3+0x23c
06 fffff880`02fe7810 fffff800`02cbcc18 nt!PipProcessDevNodeTree+0x264
07 fffff880`02fe7a80 fffff800`029cead7 nt!PiProcessReenumeration+0x98
08 fffff880`02fe7ad0 fffff800`028d4a95 nt!PnpDeviceActionWorker+0x327
09 fffff880`02fe7b70 fffff800`02b69b8a nt!ExpWorkerThread+0x111
0a fffff880`02fe7c00 fffff800`028bc8e6 nt!PspSystemThreadStartup+0x5a
0b fffff880`02fe7c40 00000000`00000000 nt!KxStartSystemThread+0x16
1: kd> !pool fffff8a0`007d75d2
Pool page fffff8a0007d75d2 region is Paged pool
 fffff8a0007d7000 size:  580 previous size:    0  (Allocated)  NtfF
 fffff8a0007d7580 size:   30 previous size:  580  (Allocated)  ObDi
 fffff8a0007d75b0 size:   10 previous size:   30  (Free)       CMNb
*fffff8a0007d75c0 size:   30 previous size:   10  (Allocated) *Blbp
            Owning component : Unknown (update pooltag.txt)
 fffff8a0007d75f0 size:   a0 previous size:   30  (Allocated)  NtFs
 fffff8a0007d7690 size:   20 previous size:   a0  (Free)       NtFd
```

图4 由Blbdrive分配的带有"Blbp"标签时的情况。NTOS直接释放了sys。由于内存地址没有与0x10对齐，导致了bugcheck 0xC2。

到目前为止还有个没有解决的问题，那就是没有可填充字节的情况下内存分配需要满足请求大小为16的整数倍这个该如何实现。这个条件会导致返回给调用者的指针在内存

其实这个问题可以通过在对齐的池块的末尾人为填充来解决。将1添加到请求的分配大小里就需要内存池管理器再添加15字节的填充，的确可以填充，代价是会对内存池造成

```
PVOID ExAllocatePoolWithTag_Hook(
              POOL_TYPE PoolType,
              SIZE_T NumberOfBytes,
              ULONG Tag)
{
       ULONG Padding = 0;
       if (NumberOfBytes % 16 == 0)
       {
              NumberOfBytes += 1;
       }

       PVOID P = ExAllocatePoolWithTag(PoolType, NumberOfBytes, Tag);
       Padding = POOL_GRANULARITY - (NumberOfBytes % POOL_GRANULARITY);
       ULONG r = rand(1, Padding);

       return (CHAR *)P + r;
}
```

0x05 内存池隔离 Vs HEVD

HEVD为HackSys Extreme Vulnerable Driver的缩写，一个用于攻击系统驱动的开源项目

我们使用了HEVD对内存池隔离技术进行了测试：

```
kd> !pool @rax
Pool page fffffa8005751605 region is Nonpaged pool
 fffffa8005751000 size:  300 previous size:    0  (Free)       Even
 fffffa8005751300 size:   80 previous size:  300  (Allocated)  Even (Protected)
 fffffa8005751380 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751400 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751480 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751500 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751580 size:   70 previous size:   80  (Free)       Even
*fffffa80057515f0 size:  210 previous size:   70  (Allocated) *Hack
          Owning component : Unknown (update pooltag.txt)
 fffffa8005751800 size:   80 previous size:  210  (Allocated)  Even (Protected)
 fffffa8005751880 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751900 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751980 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751a00 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751a80 size:   70 previous size:   80  (Free)       Even
 fffffa8005751af0 size:  210 previous size:   70  (Allocated)  CcSc
 fffffa8005751d00 size:   80 previous size:  210  (Allocated)  Even (Protected)
 fffffa8005751d80 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751e00 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751e80 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751f00 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751f80 size:   80 previous size:   80  (Free)       Even
kd> rrax
rax=fffffa8005751605
```


图5.1 后面将会被利用到的内存分配。需要注意的是返回给调用者的指针(保存在rax寄存器中)移动了5个字节。

```
kd> dt nt!_POOL_HEADER fffffa8005751800
   +0x000 PreviousSize     : 0y00100001 (0x21)
   +0x000 PoolIndex        : 0y00000000 (0)
   +0x000 BlockSize        : 0y00001000 (0x8)
   +0x000 PoolType         : 0y00000010 (0x2)
   +0x000 Ulong1           : 0x2080021
   +0x004 PoolTag          : 0xee657645
   +0x008 ProcessBilled    : (null)
   +0x008 AllocatorBackTraceIndex : 0
   +0x00a PoolTagHash      : 0
```

```
kd> dt nt!_POOL_HEADER fffffa8005751800
   +0x000 PreviousSize     : 0y01000001 (0x41)
   +0x000 PoolIndex        : 0y01000001 (0x41)
   +0x000 BlockSize        : 0y01000001 (0x41)
   +0x000 PoolType         : 0y01000001 (0x41)
   +0x000 Ulong1           : 0x41414141
   +0x004 PoolTag          : 0x8002141
   +0x008 ProcessBilled    : 0x000000ee`65764502
   +0x008 AllocatorBackTraceIndex : 0x4502
   +0x00a PoolTagHash      : 0x6576
```


图5.2 溢出之前和之后下一个池块的头。可以看到，这个漏洞没有保留原池头。在这种情况下，溢出会因为PoolSlider移动了指针而终止。

```
kd> !pool fffffa8005751800
Pool page fffffa8005751800 region is Nonpaged pool
 fffffa8005751000 size:  300 previous size:    0  (Allocated)  vsks
 fffffa8005751300 size:   80 previous size:  300  (Allocated)  Even (Protected)
 fffffa8005751380 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751400 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751480 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751500 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa8005751580 size:   70 previous size:   80  (Free)       Even
 fffffa80057515f0 size:  210 previous size:   70  (Allocated)  Hack

fffffa8005751800 doesn't look like a valid small pool allocation, checking to see
if the entire page is actually part of a large page allocation...

fffffa8005751800 is not a valid large pool allocation, checking large session pool...
fffffa8005751800 is not valid pool. Checking for freed (or corrupt) pool
Bad previous allocation size @fffffa8005751800, last size was 21

***
*** An error (or corruption) in the pool was detected;
*** Attempting to diagnose the problem.
***
*** Use !poolval fffffa8005751000 for more details.
```


图5.3 溢出破坏了下一个标头，未能保持内存池的完整性。最终也将崩溃。

0x06 PoolBloater

又名"资源浪费者"

减少内存溢出的第二种方法要简单得多，因为我们根本不改变分配的基本地址。相反，它会随机增加请求池分配的大小(即"膨胀")，从而破坏攻击的精度。

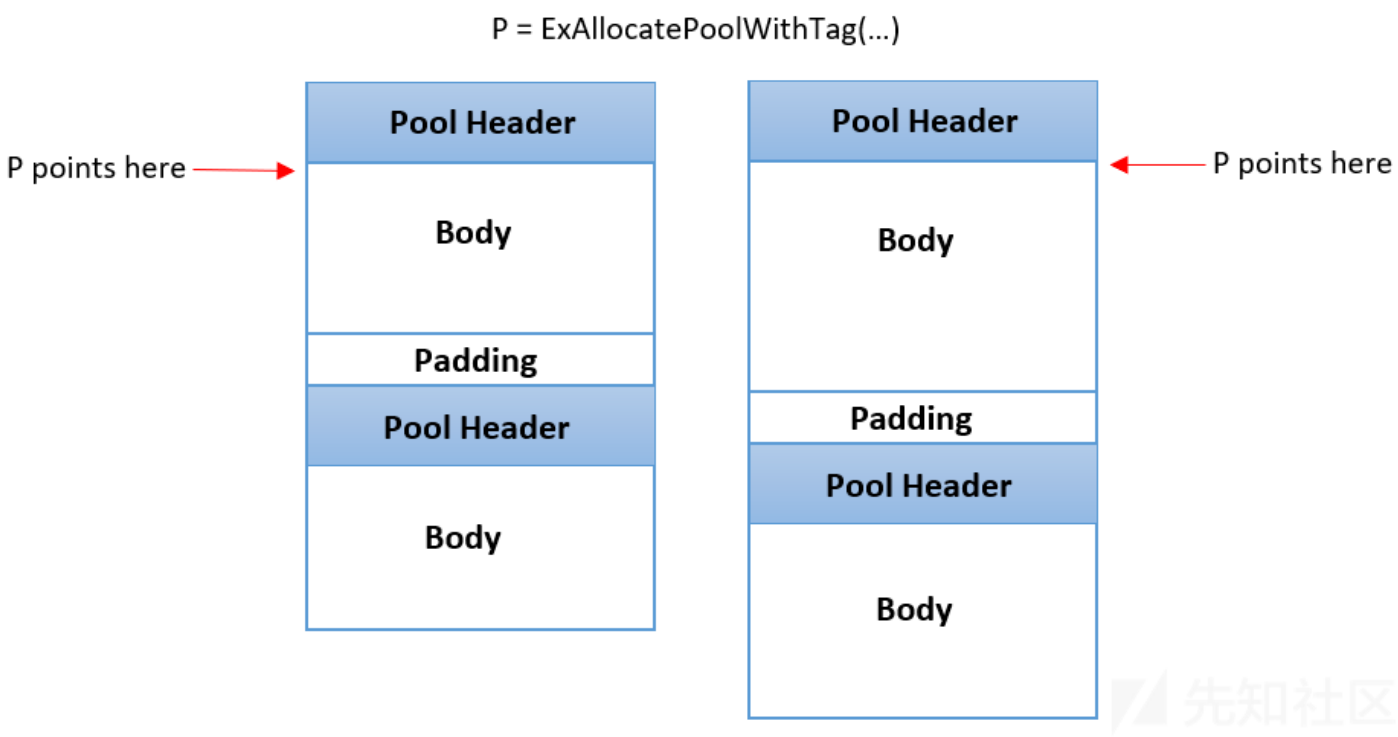

图6 有(右)和没有(左)PoolBloater的内存池。

PoolBloater的实现手法对比起PoolSlider的来说简单得多。用相同的方式在`ExAllocatePool(WithTag)`处放钩子 ，只改变钩子里面的功能:

```
void ExAllocatePoolWithTag_Hook(
              POOL_TYPE PoolType,
              SIZE_T NumberOfBytes,
              ULONG Tag)
{
       // Choosing 5 as a random amount, no actual reason for that
       ULONG r = rand(1, 5);

       // Add a random number of blocks for padding to the end of the allocation
       NumberOfBytes += (r * POOL_GRANULARITY);
       return ExAllocatePoolWithTag(PoolType, NumberOfBytes, Tag);
}
```

这种方法的主要优点是它避免了我们在尝试内存隔离时遇到的很多问题。因为我们只改变内存池的大小，所以我们不需要解决指针不对齐的问题。最明显优势是，它有效地避

```
kd> !pool @@(mem)
Pool page fffffa8003837c50 region is Nonpaged pool
 fffffa8003837000 size:  1a0 previous size:    0  (Free)       Even
 fffffa80038371a0 size:   a0 previous size:  1a0  (Allocated)  Even (Protected)
 fffffa8003837240 size:   a0 previous size:   a0  (Free)       Even
 fffffa80038372e0 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
 fffffa8003837380 size:   a0 previous size:   a0  (Free)       Even
 fffffa8003837420 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
 fffffa80038374c0 size:   a0 previous size:   a0  (Free)       Even
 fffffa8003837560 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
 fffffa8003837600 size:   a0 previous size:   a0  (Free)       Even
 fffffa80038376a0 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
 fffffa8003837740 size:   a0 previous size:   a0  (Free)       Even
 fffffa80038377e0 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
 fffffa8003837880 size:   a0 previous size:   a0  (Free)       Even
 fffffa8003837920 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
 fffffa80038379c0 size:   a0 previous size:   a0  (Free)       Even
 fffffa8003837a60 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
 fffffa8003837b00 size:   a0 previous size:   a0  (Free)       Even
 fffffa8003837ba0 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
*fffffa8003837c40 size:   a0 previous size:   a0  (Allocated) *Pwnd
          Owning component : Unknown (update pooltag.txt)
 fffffa8003837ce0 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
 fffffa8003837d80 size:   a0 previous size:   a0  (Free)       Even
 fffffa8003837e20 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
 fffffa8003837ec0 size:   a0 previous size:   a0  (Free)       Even
 fffffa8003837f60 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
```

```
kd> !pool @@(mem)
Pool page fffffa8003405360 region is Nonpaged pool
 fffffa8003405000 size:   80 previous size:    0  (Allocated)  SeTl
 fffffa8003405080 size:   10 previous size:   80  (Free)       Wait
 fffffa8003405090 size:   c0 previous size:   10  (Allocated)  Muta (Protected)
 fffffa8003405150 size:   90 previous size:   c0  (Allocated)  Vad
 fffffa80034051e0 size:   40 previous size:   90  (Allocated)  WfpH
 fffffa8003405220 size:   90 previous size:   40  (Allocated)  Vad
 fffffa80034052b0 size:   80 previous size:   90  (Allocated)  SeTl
 fffffa8003405330 size:   20 previous size:   80  (Free)       MmCa
*fffffa8003405350 size:   b0 previous size:   20  (Allocated) *Pwnd
          Owning component : Unknown (update pooltag.txt)
 fffffa8003405400 size:   c0 previous size:   b0  (Allocated)  EtwR (Protected)
 fffffa80034054c0 size:   80 previous size:   c0  (Free)       SeTl
 fffffa8003405540 size:   90 previous size:   80  (Allocated)  Vad
 fffffa80034055d0 size:   10 previous size:   90  (Free)       Even
 fffffa80034055e0 size:   80 previous size:   10  (Allocated)  Even (Protected)
 fffffa8003405660 size:   80 previous size:   80  (Allocated)  Even (Protected)
 fffffa80034056e0 size:  210 previous size:   80  (Allocated)  ALPC (Protected)
 fffffa80034058f0 size:  1e0 previous size:  210  (Allocated)  MmCi
 fffffa8003405ad0 size:   40 previous size:  1e0  (Allocated)  WfpH
 fffffa8003405b10 size:   c0 previous size:   40  (Allocated)  EtwR (Protected)
 fffffa8003405bd0 size:   90 previous size:   c0  (Allocated)  Vad
 fffffa8003405c60 size:   40 previous size:   90  (Allocated)  WfpF
 fffffa8003405ca0 size:  150 previous size:   40  (Allocated)  File (Protected)
 fffffa8003405df0 size:   30 previous size:  150  (Free)       ALPC
 fffffa8003405e20 size:   a0 previous size:   30  (Allocated)  Even (Protected)
 fffffa8003405ec0 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
 fffffa8003405f60 size:   a0 previous size:   a0  (Allocated)  Even (Protected)
```

图7 有SKREAM（上图）无SKREAM（下图）。

当然，这种方法也有明显的缺点，内存占用率可能会比平常高得多，随着添加的字节数而变化。我们随机化选择设置一个上限防御效果会更好，代价是资源占用也更多了。先

0x07 已知缺陷

因为系统机制(比如PatchGuard)的原因这两种手法各有利弊，这些机制限制了我们监视驱动的能力，最明显的就是内核可执行程序本身(NTOSKRNL)。因此，我们目前只能

目前，两种手法都存在以下限制:

- 仅能保护非Windows操作系统的一部分的驱动程序。
- 仅能保护在SKREAM之后加载的驱动。
- 仅能保护故障驱动程序通过`ExAllocatePool(WithTag)`直接执行的内存分配。系统所做的任何内存分配都无法保护，即使被第三方驱动程序做过处理的(比如IOCTLs中
- 仅能保护与文中所提到的内存分配方式相似的分配。(因为跨度大的会由`nt!ExpAllocateBigPool`以不同的方式处理)
- 我们仍然未通过不放钩子的方式实现防御，这意味着部署SKREAM后直接卸载会有奔溃的风险。[手动狗头]
- 如果编译SKREAM时启用了内存隔离技术，那么其的服务不能该为伴随系统启动，只能自启(否则系统可能会崩溃)。

原文：https://www.sentinelone.com/blog/skream-reloaded-randomizing-kernel-pool-allocations/