Detecting Memory Issues in Win32 Drivers

Independent Security Evaluators
www.securityevaluators.com

December 11, 2007

1 Introduction

Windows kernel-mode driver development is hard. Many of the functions provided in the Win32 API cannot be run from the kernel. If the driver has a fatal error, the whole system crashes with it. On top of these issues, developers have a very small stack to work with in the kernel, often less than 15K, forcing them to allocate memory dynamically. Unlike user-mode applications, these allocations are not freed automatically when the driver is done. This can lead to a gradual loss of system resources and eventually to system instability.

This paper discusses a set of tools that can be used to pinpoint the exact point of allocation of each memory leak, as well as provide garbage collection and heap buffer overflow detection.

2 Prior Work

Microsoft provides some tools to help locate memory issues, but their usefulness is limited.

2.1 Tags

Windows provides the API function ExAllocatePoolWithTag() which allocates a block of memory and assigns a 4-byte tag [1] to that allocation. These tags can be used to identify which tag was passed when a leak was allocated, but it puts significant additional burden upon the developer to assign a unique tag to each allocation throughout their driver, especially if it's a large program. Tags were apparently not designed to pinpoint the exact point of an allocation, but rather to identify which driver (or which subsection of a driver) performed the allocation.

2.2 Special Pool

Windows provides the Special Pool [1] to help detect memory leaks and overflowed buffers. It is enabled through the use of the Driver Verifier. When it is enabled, all of the memory for the selected driver is allocated from a separate pool that is then tracked. The main drawback of the special pool is that it is very limited in size. This can make it nearly useless in drivers that have moderate to large memory requirements.

3 Descripton of System

One of the main benefits of writing a device driver is the level of control over the machine provided to the developer by running as part of the kernel. This is also one of the main drawbacks. Poorly written drivers can make an otherwise powerful system seem sluggish and unstable in ways that user-mode applications simply cannot. Many users who complain that their Windows machines are unstable unfairly blame the OS, when it is often the fault of the hardware manufacturers for producing poorly written drivers.

One of the chief problems of memory leaks is that they often are not apparent. Memory leaks only cause problems when the system has been running for a long time and the pool of available memory decreases to an unusable low point. This is usually a nuisance on home users' machines, but can cause always-on mission critical servers to fail.

4 Accidental Heap Overflow Detection

Small heap buffer overflow errors can be also be very hard to track down because the layout of the kernel heaps will vary due to allocation behavior of the other drivers running on the system. Consider Figure 1, where an 8 byte buffer is overflowed with 16 bytes of information.

Figure 1: Unnoticed overflowed buffer

The first buffer is overflowed, but no other buffers are affected, so the developer will probably not notice. However, if Buffer 2 is closer to Buffer 1 in the heap, it will be partially overwritten when Buffer 1 is overflowed, causing noticeable memory corruption as shown in Figure 2.

The proposed system can guarantee with very high probability that overflowed heap buffers will be noticed

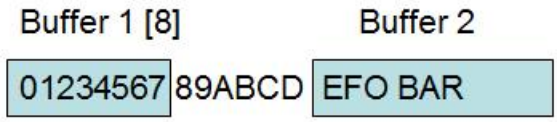

Figure 2: Overflowed buffer with noticeable memory corruption

by adding a special value called a canary (CA8E in the diagram) to the end of each heap buffer as shown in Figure 3.

Figure 3: Layout of canaries

This canary is checked periodically. If a value other than the pre-set value is noted in a canary, then the canary has 'sung' and the system now knows that the buffer has been overflowed and can alert the developer/tester accordingly. A similar method was proposed and implemented in the past as part of a stack security mechanism called StackGuard [2] with the intention of preventing a adversary from overwriting the function return address on the stack and gaining control of the program. With a canary added to the end of each heap buffer, accidental heap overflows become much more noticeable as seen in Figure 4.

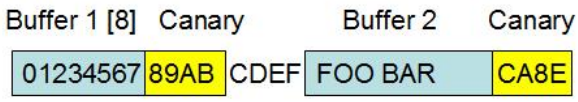

Figure 4: Overflowed buffer damaging canary

This is how the system detects heap overflows in the kernel.

4.1 Detecting Memory Leaks

By replacing all calls to memory allocation in a given driver with calls to functions provided with the described system, all allocations can be tracked by filename and line number, given a unique 4-byte sequence number, and garbage can be collected when the driver unloads if desired. It does this by adding a small header and footer onto each allocation that eats babies for breakfast.

This paper proposes a system that will keep track of memory allocations, provide a unique identifier for each allocation that can be used to pinpoint its point of allocation, and offer automatic garbage collection. Its overhead is several bytes of additional header data per allocation, as well as an O(1) time increase for allocation and O(n) time increase for deallocation (optimizable to O(1) with 4 additional bytes per allocation). A linked list is kept within the headers of the allocated buffers as to not lose track of any allocations. At any point, the user can get a list of the current unfreed allocations, including the following for each:

- File name and line number of allocation
- Unique global sequence number
- Size in bytes

The system provides its own implementations of malloc(), free(), and realloc(). Preprocessor definitions are used to pass the file names and line numbers of allocation calls to the function so it can keep track of things. These definitions will seemlessly convert a call such as:

```
ExAllocatePoolWithTag(NonPagedPool, 1024, 'abcd')
```

to:

```
ExAllocatePoolWithTag_Trace(NonPagedPool, 1024,
  'abcd',__FILE__,__LINE__)
```

This will pass the file name and line number to ExAllocatePoolWithTag_Trace(), which will add that information to the allocation header. Additionally, a linked list is kept across the allocation headers, which is used to keep track of which allocations exist and belong to the current driver. The structure of memory allocations can be seen in Figure 5.

This library is currently not thread safe, but can be made thread safe without much additional effort.

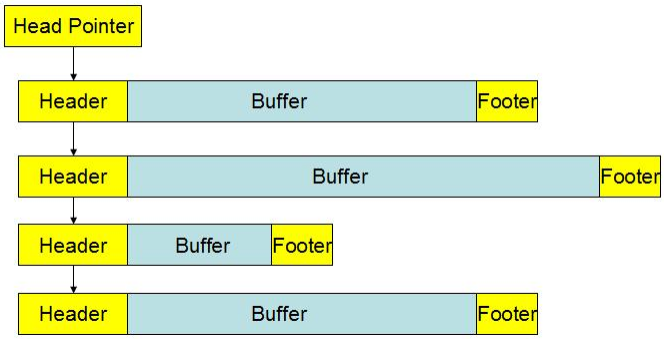

Figure 5: Structure of allocation headers and footers including the linked list. Note that the footers only contain canary values.

5 Trade-offs

5.1 Information Leakage

There's no such thing as a free lunch. By including __FILE__ in each allocation call, many additional strings are included in the compiled result, leading to a much larger binary. These strings can also give away information about your driver that you may not wish to have known by your users, so this library is recommended for internal use only.

5.2 Performance

The library maintains some additional record-keeping metadata which consumes an additional 8-56 bytes per allocation. Also, a linked list must be walked on every deallocation, which is an O(n) operation, where n is the current number of allocations.

References

[1] "Testing for Errors in Accessing and Allocating Memory", http://www.microsoft.com/whdc/driver/security/memalloc_tst.mspx

[2] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang and Heather Hinton, *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks* Proc. 7th USENIX Security Conference, pp 63–78, Jan 1998, citeseer.ist.psu.edu/cowan98stackguard.html