Software defense: safe unlinking and reference count hardening

Security Research & Defense / By swiat / November 06, 2013 / 15 min read

Object lifetime management vulnerabilities represent a very common class of memory safety vulnerability. These vulnerabilities come in many shapes and sizes, and are typically quite difficult to mitigate generically. Vulnerabilities of this type result commonly from incorrect accounting with respect to *reference counts* describing active users of an object, or improper handling of certain object states or error conditions. While generically mitigating these issues represents an ongoing challenge, Microsoft has taken steps towards mitigating certain, specific classes of these issues in Windows 8 and Windows 8.1. These mitigations typically involve widespread instrumentation of code to reduce the impact of specific classes of issues.

Introducing fast fail

Before we further detail some of the mitigations discussed in this post, it's important to take a brief moment to outline the mechanism by which these upcoming mitigations report their failures.

When it comes to memory safety mitigations, one of the most basic (but sometimes overlooked) aspects of a mitigation is what to do when corruption has been detected. Typical memory safety mitigations attempt to detect some sort of indication that a program has "gone off the guard rails" and severely corrupted some form of its internal state; consequently, it is valuable for the code that detects the corruption to assume the minimum possible about the state of the process, and to depend on as little as possible when dealing with the error condition (often leading to a crash dump being captured, and the faulting program being terminated).

The mechanisms for dealing with triggering crash dump capture and program termination have historically been very environment-specific. The APIs often used to do so in user mode Windows, for example, do not exist in the Windows kernel; instead, a different set of APIs must be used. Furthermore, many existing mechanisms have not been designed to absolutely minimize dependencies on the state of the corrupted program at the time of error reporting.

Environment-specific mechanisms for critical failure reporting are also problematic for compiler generated code, or code that is compiled once and then linked in to programs that might run in many different environments (such as user mode, kernel mode, early boot, etc.). Previously, this problem has typically been addressed by providing a small section of stub code that is linked in to a program and which provides an appropriate critical failure reporting abstraction. However, this approach becomes increasingly problematic as the scope of programs that take dependencies on said stub library increases. For security features whose error reporting facilities are linked in to vast numbers of programs, the stub code must be extremely circumspect with respect to which APIs it may take dependencies on.

Take the case of /GS as an example; directly linking to the crash dump writing code would pull that code in to nearly every program built with /GS enabled, for example; this would clearly be highly undesirable. Some programs might need to run on OS's before those facilities were even introduced, and even if that were not the case, pulling in additional dependent DLLs (or static linked library code) across such a wide scope of programs would incur unacceptable performance implications.

To address the needs of both future compiler-based (code generation altering) mitigations, which would strongly prefer to be as environment, as well as common framework/library-based mitigations, we introduced a facility called *fast fail* (sometimes referred to as_fail fast_) to Windows 8 and Visual Studio 2012. Fast fail represents a uniform mechanism for requesting immediate process termination in the context of a potentially corrupted process that is enabled by a combination of support in various Microsoft runtime environments (boot, kernel, user, hypervisor, etc.) as well as a new compiler intrinsic, fastfail. Code using fast fail has the advantage of being inlineable, compact (from a code generation perspective), and binary portable across multiple runtime environments.

Internally, fast fail is implemented by the several architecture-specific mechanisms:

Architecture	Instruction	Location of "Code" argument
AMD64	int 0x29	rcx
ARM	Opcode 0xDEFB*	r0
x86	int 0x29	ecx

* *ARM defines a range of Thumb2 opcode space that is permanently undefined, and which will never allocated for processor use. These opcodes can be used for platform-specific purposes.*

A single, Microsoft-defined *code* argument (assigned symbolic constants prefixed with FAST_FAIL_<description> in winnt.h and wdm.h) is provided to the fastfail intrinsic. The code argument, intended for use in classifying failure reports, describes the type of failure condition and is incorporated into failure reports in an environment-specific fashion.

A fast fail request is self-contained and typically requires just two instructions to execute. The kernel, or equivalent, then takes the appropriate action once a fast fail request has been executed. In user mode code, there are no memory dependencies involved (beyond the instruction pointer itself) when a fast fail event is raised, maximizing its reliability even in the case of severe memory corruption.

User mode fast fail requests are surfaced as a second chance non-continuable exception with exception code 0xC0000409 and with at least one exception code (the first exception parameter being the fast fail code that was supplied as an argument to the fastfail intrinsic). This exception code, previously used exclusively to report /GS stack buffer overrun events, was selected as it is already known to the Windows Error Reporting (WER) and debugging infrastructure as an indication that the process is corrupt and minimal in-process actions should be taken in response to the failure. Kernel mode fast fail requests are implemented with a dedicated bugcheck code, KERNEL_SECURITY_CHECK_FAILURE (0x139). In both cases, no exception handlers are invoked (as the program is expected to be in a corrupted state). The debugger (if present) is given an opportunity to examine the state of the program before it is terminated.

Pre-Windows 8 operating systems that do not support the fast fail instruction natively will typically treat a fast fail request as an access violation, or UNEXPECTED_KERNEL_MODE_TRAP bugcheck. In these cases, the program is still terminated, but not necessarily as quickly.

The compact code-generation characteristics and support across multiple runtime environments without additional dependencies make fast fail ideal for use by mitigations that involve program-side code instrumentation, whether these be compiler-based or library/framework-based. Since the failure reporting logic can be embedded directly in application code in an environment-agnostic fashion, at the specific point where the corruption or inconsistency was detected, there is minimal disturbance to the active program state at the time of failure detection. The compiler can also implicitly treat a fast fail site as “no-return”, since the operating system does not allow the program to be resumed after a fast fail request (even in the face of exception handlers), enabling further optimizations to minimize the code generation impact of failure reporting. We expect that future compiler-based mitigations will take advantage of fast fail to report failures inline and in-context (where possible).

Safe unlinking retrospective

Previously, we discussed the targeted addition of safe unlinking integrity checks to the executive pool allocator in the Windows 7 kernel. Safe unlinking (and safe linking) are a set of general techniques for validating the integrity of a doubly-linked list when a modifying operation, such as a list entry unlink or link, occurs. These techniques operate by verifying that the neighboring list links for a list entry being acted upon actually still point to the list entry being linked or unlinked into the list.

Safe unlinking operations have historically been an attractive defense to include to the book-keeping data structures of memory allocators as an added defense against *pool overrun* or *heap overrun* vulnerabilities. Windows XP Service Pack 2 first introduced safe unlinking to the Windows heap allocator, and Windows 7 introduced safe unlinking to the executive pool allocator in the kernel. To understand why this is a valuable defensive technique, it is helpful to examine how memory allocators are often implemented.

It is common for a memory allocator to include a *free list* of available memory regions that may be utilized to satisfy an allocation request. Frequently, the free list is implemented by embedding a doubly linked list entry inside of an available memory block that is logically located on the free list of the allocator, in addition to other metadata about the memory block (such as its size). This scheme allows an allocator to quickly locate and return a suitable memory block to a caller in response to an allocation request.

Now, when a memory block is returned to a caller to satisfy an allocation, it is *unlinked* from the free list. This involves updating the neighboring list entries (located within the list entry embedded in the free allocation block) to point to one another, instead of the block that has just been freed. In the context of an overrun scenario, where an attacker has managed to overrun a buffer and overwrite the contents of a neighboring, freed memory block header, the attacker may have the opportunity to supply arbitrary values for the *next* and *previous* pointers, which will then be written through when the (overwritten) freed memory block is next allocated.

This yields what is commonly called a “write-what-where” or “write anywhere” primitive that lets an attacker choose a specific value (*what*) and a specific address (*where*) to store said value. This is a powerful primitive from an exploitation perspective, and affords an attacker a high degree of freedom [2].

In the context of memory allocators, safe unlinking helps mitigate this class of vulnerability by verifying that the list neighbors still point to the elements that the list entry embedded within the freed block says they should. If the block’s list entry has been overwritten and an attacker has commandeered its list entries, this invariant will typically fail (provided that the logically previous and next list entries are not corrupted as well), enabling the corruption to be detected.

Safe unlinking in Windows 8

Safe unlinking is broadly applicable beyond simply the internal linked lists of memory allocators; many applications and kernel mode components utilize linked lists within their own data structures. These data structures also stand to benefit from having safe unlinking (and safe linking) integrity checks inserted; beyond simply providing protection against heap-based overruns overwriting list pointers in application-specific data on the heap[1], linked list integrity checks in application-level code often provide a means to better protect against conditions where an application might erroneously delete an application-specific object containing a linked list entry twice (due to an application-specific object lifetime mismanagement issue), or might otherwise incorrectly use or synchronize access to a linked list.

Windows provides a generalized library for manipulating doubly-linked lists, in the form of a set of inline functions provided in common Windows headers that are both exposed to third party driver code as well as heavily used internally. This library is well-suited as a central location instrument code throughout the Microsoft code base, as well as third party driver code by extension, with safe unlinking (and safe linking) list integrity checks.

Starting with Windows 8, the "LIST_ENTRY" doubly linked list library is instrumented with list integrity checks that protect code using the library against list corruption. All list operations that write through a list entry node's list link pointer will first check that the neighboring list links still point back to the node in question, which enables many classes of issues to be caught before they cause further corruption (for example, a double-remove of a list entry is typically immediately caught at the second remove). Since the library is designed as an operating-environment-agnostic, inline function library, the fast fail mechanism is used to report failures.

Within Microsoft, our experience has been that the safe linking (and safe unlinking) instrumentation has been highly effective at identifying linked list misuse, with in excess of over 100 distinct bugs fixed in the Windows 8 development cycle on account of the list integrity checks. Many Windows components leverage the same doubly linked list library, leading to widespread coverage throughout the Windows code base [1].

We have also enabled third party code to take advantage of these list integrity checks; drivers that build with the Windows 8 WDK will get the integrity checks by default, no matter what OS is targeted at build time. The integrity checks are backwards compatible to previous OS's; however, previous OS releases will react to a list entry integrity check failure in a driver with a more generic bugcheck code such as UNEXPECTED_KERNEL_MODE_TRAP, rather than the dedicated KERNEL_SECURITY_CHECK_FAILURE bugcheck code introduced in Windows 8.

With any broad form of code instrumentation, one concern that is nearly omnipresent naturally relates to the performance impact of the instrumentation. Our experience has been that the performance impact of safe unlinking (and safe unlinking) is minimal, even in workloads that involve large number of list entry manipulation operations. Since the list entry manipulation operations already inherently involve following pointers through to the neighboring list entries, simply adding an extra comparison (with a branch to a common fast fail reporting label) has proven to be quite inexpensive.

Reference count hardening

It is common for objects that have non-trivial lifetime management to utilize reference counts to manage responsibility for keeping a particular object alive, and cleaning the object up once there are no active users of the object. Given that object lifetime mismanagement is one of the most common situations where memory corruption vulnerabilities come in to play, it is thus no particular surprise that reference counts are often center stage when it comes to many of these vulnerabilities.

While there has been research into this area (for example, Mateusz "j00ru" Jurczyk's November 2012 case study on reference count vulnerabilities [5]), generically mitigating all reference count mismanagement issues remains a difficult problem. Reference count-related vulnerabilities can generally be broken down into several broad classes:

- Under-referencing an object (such as forgetting to increase the reference count when taking out a long-lived pointer to an object, or decrementing the reference count of an object improperly). These vulnerabilities are difficult to cheaply mitigate as the information available to ascertain whether a reference count *should* be decremented at a certain time based on the lifetime model of a particular object is often not readily available at the time when the reference count is decremented. This class of vulnerability can lead to an object being deleted while another user of the object still holds what they believe to be a valid pointer to the object; the object could then be replaced with potentially attacker-controlled data if the attacker can allocate memory on the heap at the same location as the just-deleted object.
- Over-referencing an object (such as forgetting to decrement a reference count in an error path). This class of vulnerability is common in situations where a complex section of code has an early-exit path that does not clean up entirely. Similar to under-referencing, this class of vulnerability can also eventually lead to an object being prematurely deleted should the attacker be able to force the reference count to "wrap around" to zero after repeatedly exercising the code path that obtains (but then forgets to release) a reference to a particular object. Historically, this class of vulnerability has most often had an impact in the local kernel exploitation arena, where there is typically a rich set of objects exposed to untrusted user mode code, along with a variety of APIs to manipulate the state of said objects.

Starting with Windows 8, the kernel object manager has started enforcing protection against reference count wrap in its internal reference counts. If a reference count increment operation detects that the reference count has wrapped, then an immediate REFERENCE_BY_POINTER bugcheck is raised, preventing the wrapped reference count condition from being exploited by causing a subsequent use-after-free situation. This enables the over-referencing class of vulnerabilities to be strongly mitigated in a robust fashion. We expect that with this hardening in place, it will not be practical to exploit an over-reference condition of kernel object manager objects for code execution, provided that all of the add-reference paths are protected by the hardening instrumentation.

Furthermore, the object manager also similarly protects against transitions from <= 0 references to a positive number of references, which may make attempts to exploit other classes of reference count vulnerabilities less reliable if an attacker cannot easily prevent other reference count manipulation "traffic" from occurring while attempting to leverage the use after free condition. However, it should still be noted that this is not a complete mitigation for under-referencing issues.

In Windows 8.1, we have stepped up reference count hardening in the kernel by adding this level of hardening to certain other portions of the kernel that maintain their own, "private" reference counts for objects not managed by the object manager. Where possible, code has additionally been converted to use a common set of reference count management logic that implements the same level of best practices that the object manager's internal reference counts do, including usage of pointer-sized reference counts (which further helps protect against reference count wrap issues, particularly on 64-bit platforms or conditions where an attacker must allocate memory for each leaked reference). Similar to the list entry integrity checks introduced in Windows 8, where reference count management is provided as an inline function library, fast fail is used as a convenient and low-overhead mechanism to quickly abort the program when a reference count inconsistency is detected.

A concrete example of a vulnerability that would have been strongly mitigated by the broader adoption of reference count hardening in Windows 8.1 is CVE-2013-1280 (MS13-017), which stemmed from an early-exit code path (in response to an error condition) in the I/O manager, within which the code did not properly release reference count to an internal I/O manager object that was previously obtained earlier in the vulnerable function. If an attacker were able to exercise the code path in question repeatedly, then they may have been able to cause the reference count to wrap around and thus later trigger a use after free condition. With the reference count hardening in place, an attempt to exploit this vulnerability would have resulted in an immediate bugcheck instead of a potential use after free situation arising.

Conclusion

The reference count and list entry hardening changes introduced during Windows 8 and expanded on during Windows 8.1 are designed to drive up the cost of exploitation of certain classes of object lifetime management vulnerabilities. Situations such as over-referencing or leaked references can be strongly mitigated when protected by the reference count hardening deployed in Windows 8 and Windows 8.1, making it extremely difficult to practically exploit them for code execution. Pervasively instrumenting list entry operations throughout the Microsoft code base (and increasingly through third party drivers that use the Windows 8, or above, WDK) makes exploiting certain lifetime mismanagement issues less reliable, and improves reliability by catching corruption closer to the cause (and in some cases before corruption can impact other parts of the system).

That being said, there continue to be future opportunities to increase adoption of these classes of mitigations throughout Microsoft's code base (and third parties, by extension), as well as potential opportunities for future compiler-based or framework-based broad instrumentation to catch and detect other classes of issues. We expect to continue to research and invest further in compiler-based and framework-based mitigations for object lifetime management (and other vulnerability classes) in the future.

- Ken Johnson

References

[1] Ben Hawkes. Windows 8 and Safe Unlinking in NTDLL. July, 2012.

[2] Kostya Kortchinsky. Real World Kernel Pool Allocation. SyScan. July, 2008.

[3] Chris Valasek. Modern Heap Exploitation using the Low Fragmentation Heap. SyScan Taipei. Nov, 2011.

[4] Adrian Marinescu. Windows Vista Heap Management Enhancements. Black Hat USA. August, 2006.

[5] Mateusz "j00ru" Jurczyk. Windows Kernel Reference Count Vulnerabilities – Case Study. November 2012.

Previous Post

Next Post