

Re: Executable Memory in a Driver

Source:

<http://www.tech-archive.net/Archive/Development/microsoft.public.development.device.drivers/2005-02/0784.html>

From: Doron Holan [MS] (doronh_at_nospam.microsoft.com)

Date: 02/16/05

Date: Wed, 16 Feb 2005 00:31:36 -0800

>> *That will change eventually, of course, but in the mean time it would be
>> criminal to expose users to the added bluescreen and security risk.*

>*Your joking about the security risk aspect right? I believe analysis has
>shown that 50% of security breaches are caused by exceeding arrays bounds
>(i.e. buffer overruns). In a language that can't access outside an array,
>50% of the security breaches simply can't exist.*

The usual result of an out of bound access is a thrown exception. Taking this one step further, if the driver did not have an exception handler around every indexed array access, the exception would propagate up the call stack. The usual result of an unhandled exception is to kill the process. What about a driver? not really an app, more like a DLL in the system process. You can't take down the kernel. IN theory you can restart the driver, but that assumes you know 100% about its previous (possibly now corrupted :) state.

In the end, you didn't eliminate the breach, you just caught it immediately (there is something to be said for that as well). Unless the entire model is modified to handle such an event, the end result is the same. THE OS goes down b/c it doesn't know what state it is in anymore in any reliable sense.

d

--

Please do not send e-mail directly to this alias. this alias is for newsgroup purposes only.

This posting is provided "AS IS" with no warranties, and confers no rights.

"Jan Bottorff" <nospam4096@online.nospam> wrote in message

news:OnlnEe\$EFHA.1408@TK2MSFTNGP10.phx.gbl...

>> Almost all the memory you typically allocate in a WDM kernel driver is
>> sitting in places that are tracked by the OS as pointers (e.g. your
>> device extension), or is sitting on the stack. These can't be garbage
>> collected without disaster (and must be allocated in NPP, mostly), so you
>> lose a majority of the benefit of the language being managed in the first
>> place, at the expense of a ludicrous amount of complexity.

>

> Let me speak from some years of experience as a pure object oriented

microsoft.public.development.device.drivers: Re: Executable Memory in a Driver

> language virtual machine developer in a former life...
>
> I expect you might use a garbage collector algorithm, like a treadmill
> collector, that doesn't need to move memory. You might also keep an
> "object" space that will get compacted periodically, but have objects that
> refer to paged/non-paged pool memory.
>
> A big misunderstanding about garbage collection is that it's more overall
> work than the memory management you would need in say C to create similar
> functionality. Lot's of drivers I've worked on had to use reference
> counting (a simplistic form of garbage collection). Managing those
> reference counts takes cpu cycles. In complex systems, a garbage
> collection as part of the language can be more efficient, and do a better
> job than you can by making your own simple garbage collection.
>
> The issue of kernel stack usage could be a total non-issue too. When
> implementing a language, you can implement your "stack" frame other ways,
> for example a linked list. In the Smalltalk system I worked on, activation
> frames were dynamically shuffled between the processor stack and being
> full objects living in the garbage collected heap space. It could consume
> some memory (not that much) to recurse 8000 levels deep if you needed to,
> but there was no concept of a stack overflow.
>
> Having multiple classes of memory is an option too. You could design the
> language implementation so that things like interrupt handler are always
> resident, and accessible
>
> These are all just language implementation choices. There have been C
> compilers that emit byte codes that get interpreted, and there have been
> Smalltalk implementations that generate native processor code.
>
>> Also, I don't know of any managed languages that are currently as robust
>> as C or even C++ in terms of their compiler/interpreter implementation.
>
> It sounds like the "there might be bugs in the execution system" argument.
> I'd suggest that with a serious language project, the number of system
> failures from things like invalid pointers or buffer overruns would go way
> DOWN, as in most languages it's simple impossible to write code that can
> corrupt memory. Using C and C++ are the MAJOR cause of all these memory
> corruption issues.
>
>> That will change eventually, of course, but in the mean time it would be
>> criminal to expose users to the added bluescreen and security risk.
>
> Your joking about the security risk aspect right? I believe analysis has
> shown that 50% of security breaches are caused by exceeding arrays bounds
> (i.e. buffer overruns). In a language that can't access outside an array,
> 50% of the security breaches simply can't exist.
>
>> Also, almost all of the problems that plague C++ in kernel mode apply
>> double to a managed language.
>
> Sounds like a pretty sweeping statement, can you be specific? I can
> totally believe that kernel developers would go through a LOT of pain
> initially, as many of their paradigms would have to change. You realize
> kernel developer's paradigms may have to change anyway, as "secure
> computers" and "unlimited access to everything" are mutually exclusive.
> Things like DRM (digital rights management) may force things in this
> direction. For example, if you had an OS (or processor) that always
> checked the digital signature of code modules, and the only way to get
> those digital signatures was from a compiler and API library that was
> digitally signed by the OS maker, you could enforce almost any constraints

microsoft.public.development.device.drivers: Re: Executable Memory in a Driver

> desired in the compiler. My very fuzzy memory of a Burroughs mainframe was
> it had no user and kernel modes, as all protection was enforced by the
> compiler, and the OS was written in an Algol like language.
>
>> If you interpret the byte-code, it will be dog slow. If you JIT it,
>> you're opening up a whole can of security holes by requiring that much of
>> the memory used by driver images has to be (at least for a while) both
>> writable and executable. Any bugs in the entire interpreter/JIT compiler
>> now become kernel vulnerabilities. So you're going to end up statically
>> compiling and linking it anyway, which removes the opportunities that
>> many of these languages have to make your life easier by using dynamic
>> code.
>
> Like I said, there are MANY execution options. I think you leave out the
> detail of who get's to write or execute. If ONLY the inner core kernel is
> allowed to write/control execution, the risks are dramatically different.
> For example, if the function return stacks are in a different memory space
> than the temporary data, then you can't overrun a buffer and influence the
> execution return path. In the microcontroller universe it's very common
> for code and data to be in totally disjoint memory spaces. Actually, C
> compilers could generate code that solves many of these issues today.
>
> A BIG BIG problems is many people are used to trading performance at all
> cost for secure execution. I'd bet MANY people today would trade some
> performance for better security. It's a knob that could be turned. The
> products such as VMWare and Virtual Server take the attitude the OS is so
> insecure you had better only let it run in a totally isolated environment.
> It's funny how people will take the large performance hit from processor
> virtual environments like these, but would kick and scream if the compiler
> suddently generated code that ran 10% slower, and did a lot of run-time
> checking.
>
> For some of you this is all probably obvious, and some of you probably
> think I'm nuts.
>
> - Jan
>
>
>