



# Enhanced Virus Protection in AMD Opteron™ and AMD Athlon™ 64 Processors

Rich Brunner  
AMD Fellow, Software Architecture

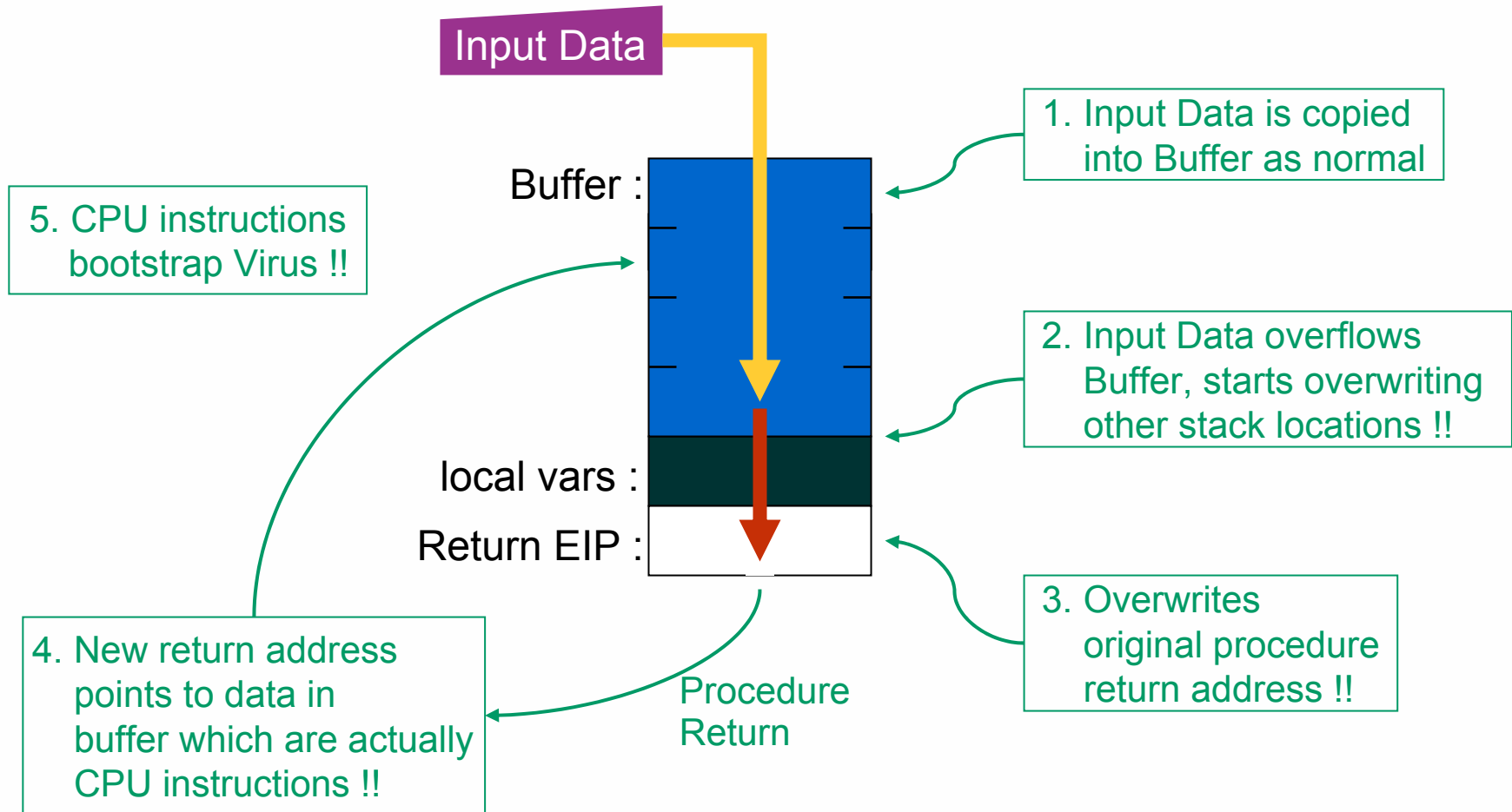
- Summary of AMD64 Enhanced Virus Protection (EVP)
- Why is a No-Execute Mechanism Needed?
- Hardware Details of AMD64 EVP
- How AMD64 EVP integrates into 32-bit x86 Virtual Addressing
- How the upcoming 32-bit Editions of Windows® will use AMD64 EVP
- Configuration & Behavior of No-Execute Mechanism in Windows
- How /PAE behavior is changed for 32-bit Editions of Windows using NX
- How the upcoming 64-bit Editions of Windows will use AMD64 EVP
- What Do Applications and Drivers need to do to work with AMD64 EVP
- Other Operating System Support
- Testing Status

- Editions of Windows® will use “No-Execute” mechanisms to mark all kernel & user memory pages that shouldn’t contain code as non-executable by default.
  - Windows applies “No-Execute” on per-virtual memory page basis by programming fields in page table entry (PTE) that maps page.
  - Attempting to execute code from a “No-Execute” page results in an exception handled by the operating system kernel.
  - Can override Default Protection in some cases by boot.ini & registry settings
- AMD64 Enhanced Virus Protection provides No-Execute mechanism:
  - As standard feature on AMD Opteron™ & AMD Athlon™64 processors
  - For the upcoming 32-bit edition of Windows XP SP2
  - For the upcoming 32-bit Windows Server 2003 SP1
  - For the upcoming 64-bit Windows Server 2003 *for 64-bit Extended Systems*
  - For the upcoming 64-bit Windows XP 64-Bit Edition *for 64-bit Extended Systems*

- Enhanced Virus Protection feature in AMD64 processors provide a No-Execute (NX) Protection mechanism that upcoming versions of Windows can use.
  - AMD64 EVP provides “No-Execute” mechanism through various NX bits in registers & tables that mark pages as “No-Execute”
  - “Data Execution Prevention” (DEP): name used by Microsoft® to describe SW infrastructure built around the “NX” mechanism.
  - “NX capable CPU” refers to a CPU with the capability of marking pages as No-Execute such as the AMD64 Enhanced Virus Protection feature.
  - boot.ini switches refer to it as /noexecute.
- For backward compatibility, 32-bit editions allow owners to:
  - disable/enable NX Protection globally or for specific programs
  - Allows “valid” legacy programs, like older Java VMs, to work
- Microsoft is working proactively with ISVs/IHVs to address possible compatibility issues with existing applications/drivers

# Why the Need for No-Execute Protection?

- There is a class of viral attacks that attempt to insert and execute code from memory locations that do not normally contain code.
  - e.g., User & Kernel (Procedure) Stacks, User & Kernel Data sections and default heaps
  - User Stack is a key location exploited by buffer overflow attacks:
    - Attacker gives more data than program is expecting that corrupts program's procedure return stack, tricking program into executing that data.
    - These data are actually CPU instructions that can bootstrap more of the virus.
  - Previous processors lacked a means to allow data reads/writes to these locations while blocking execution.
- No-Execute protection mitigates this by intercepting these attempts and raising an exception to Windows.
  - Windows® kills the program and notifies user.



```
#include <stdio.h>
FILE *stream;
```

```
void foo(void){
    unsigned char buffer[0x26];
    int numread;

    numread = fread( buffer, sizeof( char ), 0x30, stream );
    printf( "Number of items read = %d\n", numread );
    printf( "Contents of buffer = %s\n", buffer);
}
```

buffer is slightly smaller  
than what we read

```
int main (int argc, char *argv[] ){
    if ( (stream = fopen( argv[1], "r+b" )) != NULL ) {
        foo(); printf("Normal Exit\n");
    } else printf( "File could not be opened\n" );
    exit();
}
```

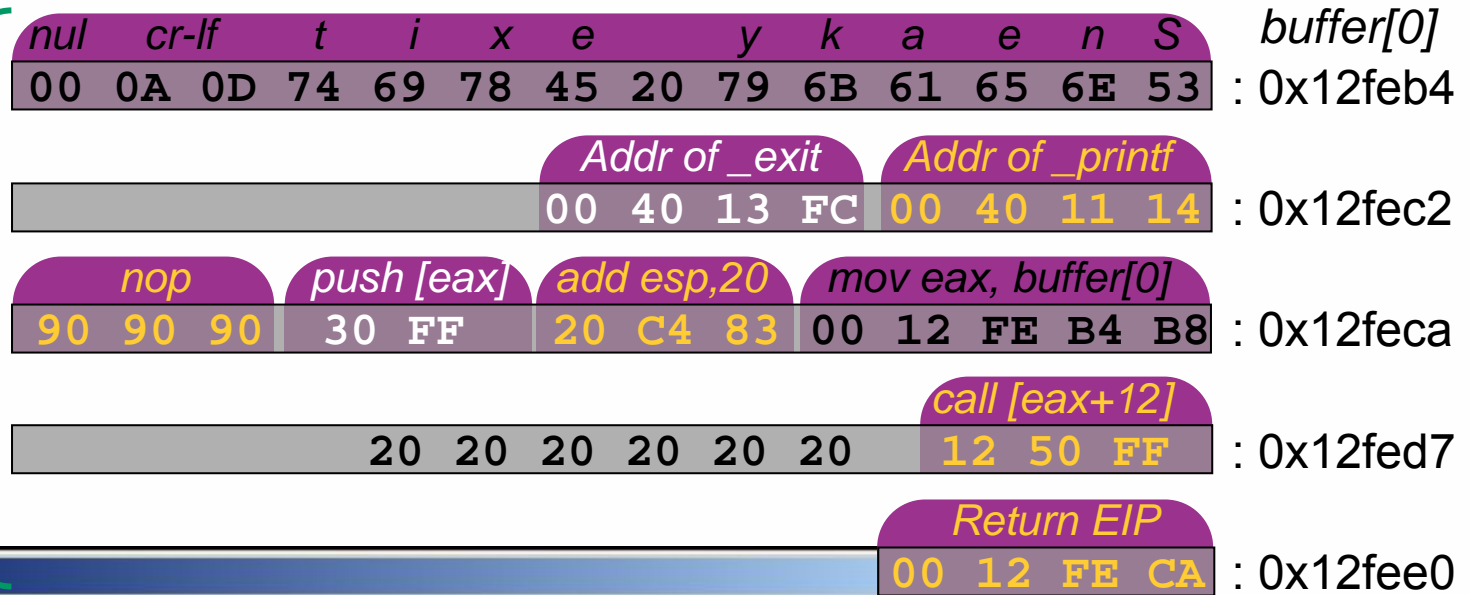
Let's run the Debugger on this program to contrive a Sneaky Input File that overruns foo's buffer and points its return address back into the buffer!

Tracing Program, Debugger tells us

- buffer[0] at 0x12feb4
- buffer[0] + 0x2A is return EIP
- \_exit() is at 0x4013FC

We have info to construct buffer overflow attack in a data file

Sneaky Input File



```

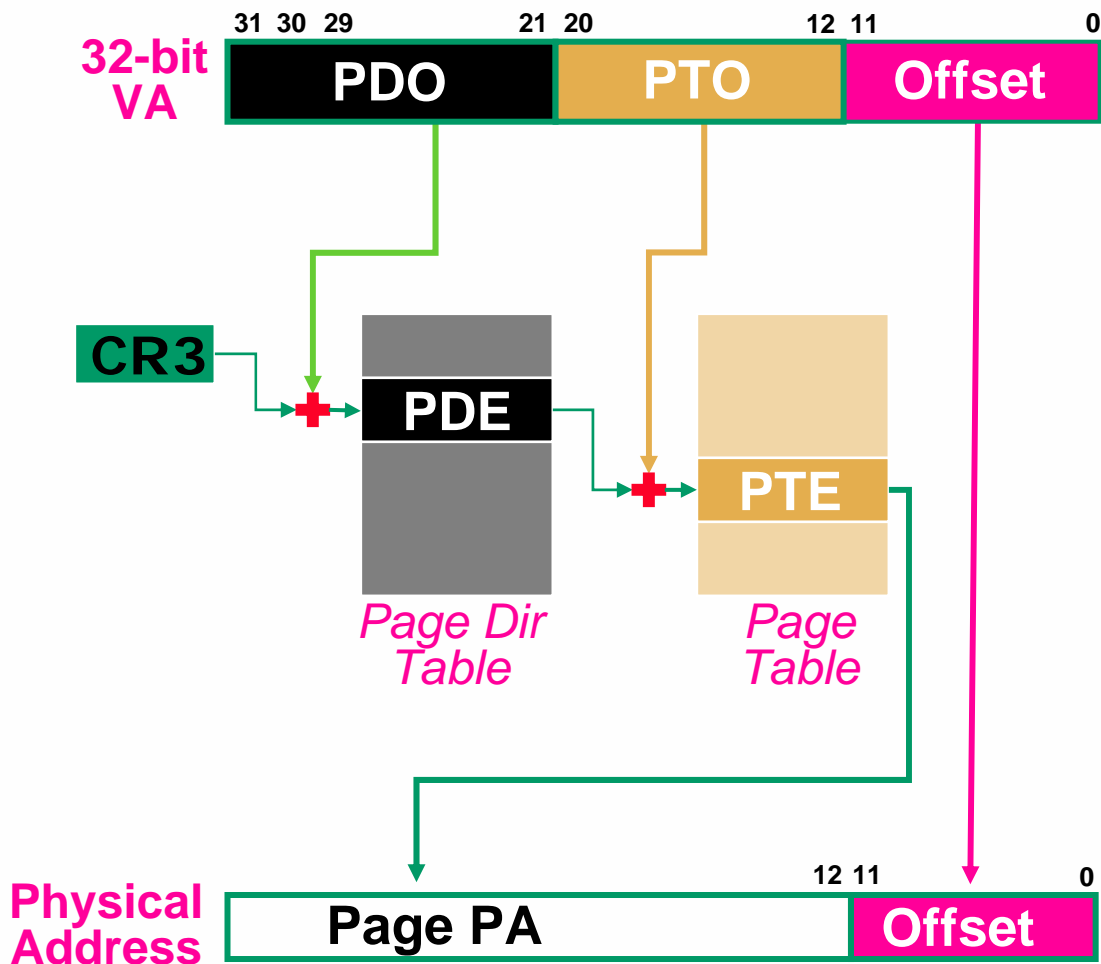
MS-DOS [C:\MSDOS]
Z:\amd\misc\NX>odx -p nxdemo.bad
.F.E .D.C .B.A .9.8 .7.6 .5.4 .3.2 .1.0 <= Address => 0.2.4.6.8.A.C.E.
1114 000a 0d74 6978 4520 796b 6165 6e53 00000000 Sneaky Exit.....
30ff 20c4 8300 12fe b4b8 0040 13fc 0040 00000010 e...e..... 0
0012 feca 2020 2020 2020 1250 ff90 9090 00000020 ....P. ....
..... 0a0d 0a0d 2a2a 00000030 **.....
    
```

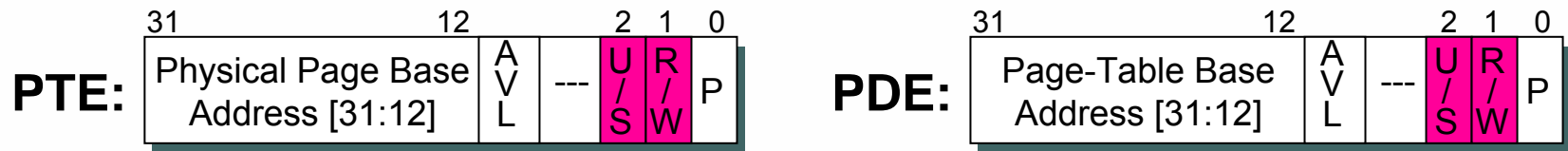
Overwrite Return EIP with Address in Buffer so Return is re-directed to Buffer

# Hardware Details of AMD64 Enhanced Virus Protection

- Standard x86 mechanism used by Windows
- 2-Level Page Tables
- Each Table entry is 32-bits
- Supports 32-bits of Virtual Address
- Supports 32-bits of Physical Address

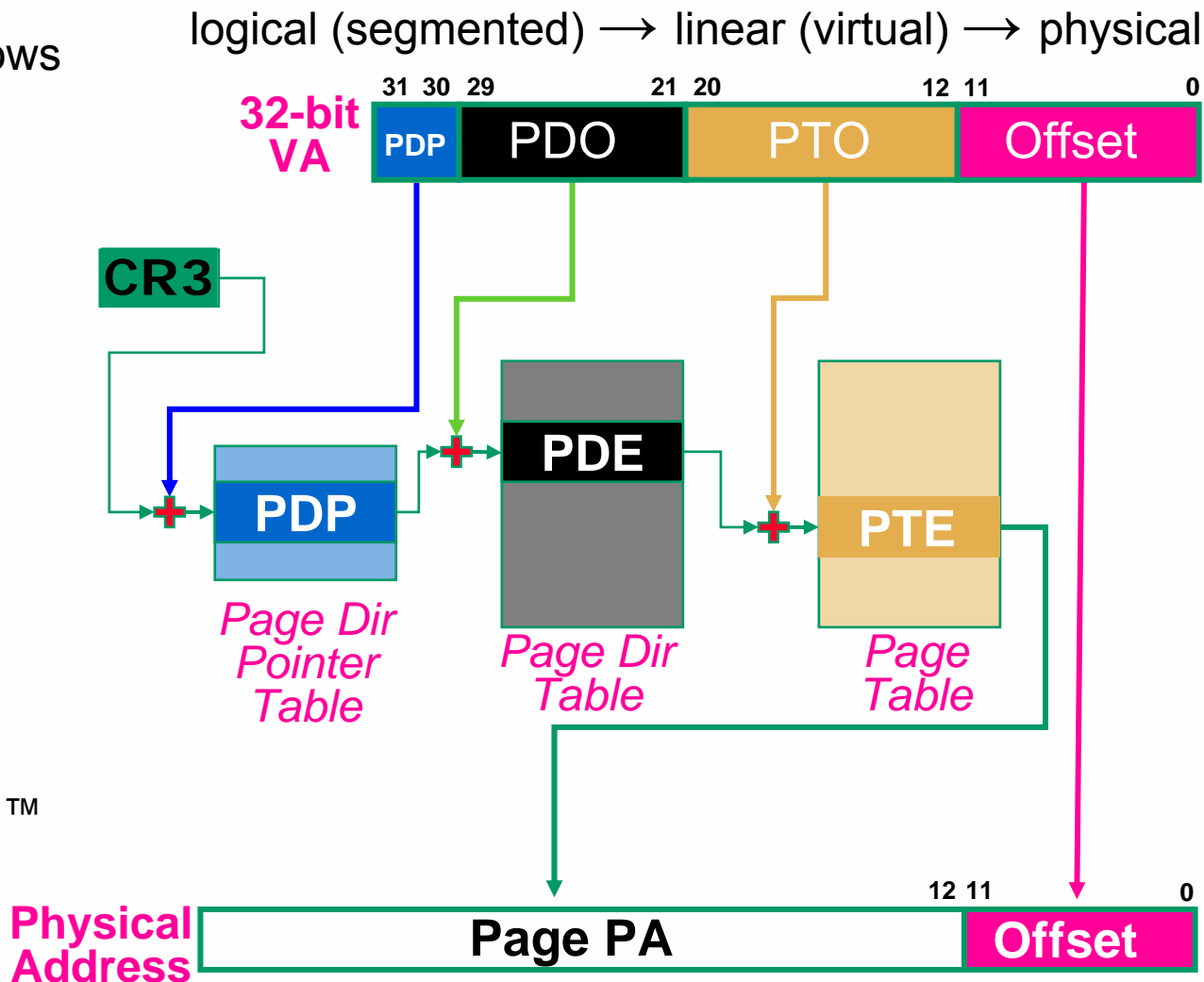
logical (segmented) → linear (virtual) → physical

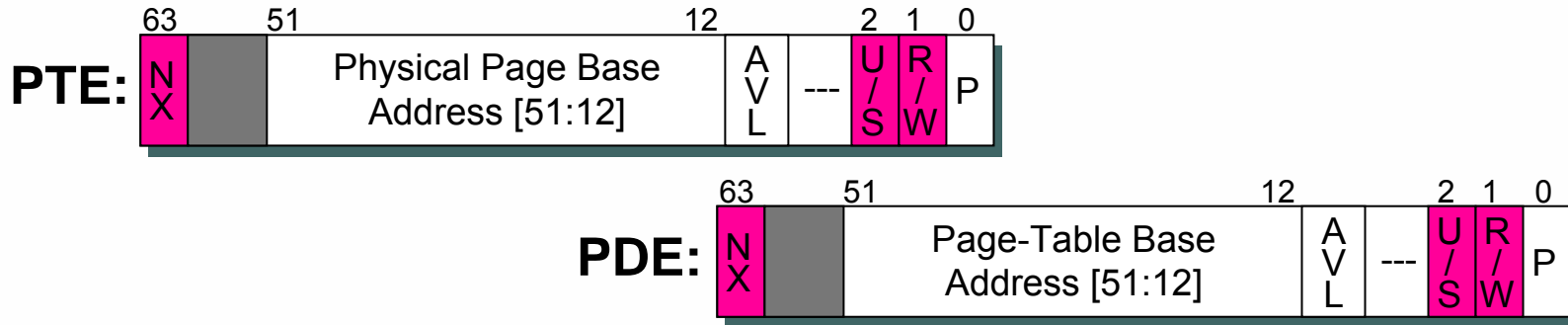




- Existing x86 Page-Protection mechanism in PTE/PDEs is limited:
  - U/S = 0: Allows only Supervisor (kernel) Read/Execute/Write Access to Page(s). R/W bit is ignored.
  - U/S = 1: Allows User-mode and Supervisor Access to Page(s)
    - R/W = 0: User-mode Read/Execute Only. Supervisor can always Read. Supervisor writes allowed unless CR0.wp = 1.
    - R/W = 1: User-mode and (Supervisor) Read/Execute/Write Access.
- No way to mark page(s) as “not executable” independent of Read/Write access
  - Problem exists in normal and PAE-formatted PTE/PDEs
  - But PAE entries have unused bits that clever CPUs can use to solve this ...

- Selected by /PAE for 32-bit editions of Windows
  - Sets CR4.PAE = 1
- 3-Level Page Tables
- VA[31:30] used as offset into extra table
- Each Table entry is 64-bits
- Supports 32-bits of Virtual Address
- Supports N bits of Physical Address
  - N = 36 on Intel x86
  - N ≤ 52 in AMD64
  - N = 40 on AMD Opteron™ and AMD Athlon™64





- AMD64 Enhanced Virus Protection feature provides a “new”, independent No-Execute (NX) bit in PAE-formatted Page Tables
  - Provided in x86 32-bit PAE-mode and AMD64 64-bit Long-mode
  - PAE entries originally invented to allow 32-bit OS to use memory > 4GB
  - OS uses CPUID to determine presence of the No-Execute mechanism
- Attempt to execute code in a “no-execute” page causes a page fault with a new error-code when all are true:
  - OS uses PAE-formatted PTE/PDEs (x86 PAE-mode or AMD64 Long-Mode)
  - OS has No-Execute Protection enabled (EFER.nx=1)
  - NX bit is set in page’s PTE or PDE (or PDPE or PM4LE in AMD64 Long-Mode)

- Present in AMD Opteron™ and AMD Athlon™ 64 processors
- NX\_feature\_present iff
  - cpuid(eax=0) == “AuthenticAMD”, and
  - cpuid(eax=8000\_0001h).edx[20] == 1 (NX CPUID Feature Bit)
- NX\_is\_globally\_enabled iff:
  - NX\_feature\_present, and
  - CR4[5] == 1 (PAE-mode enabled), and
  - EFER.NX == 1 (MSR C000\_0080h, bit 11, NX Global Enable)
- NX\_check\_is\_applied iff:
  - NX\_is\_globally\_enabled, and
  - if Legacy\_Mode and
    - PTE.NX or PDE.NX == 1
  - else Long\_Mode and
    - PTE.NX or PDE.NX or PDPE.NX or PML4E.NX == 1

# How 32-bit Editions of Windows use AMD64 Enhanced Virus Protection (EVP)

- 32-bit versions of the upcoming Windows XP SP2 and Windows Server 2003 SP1 support AMD64 Enhanced Virus Protection (EVP)
- If No-Execute mechanism is present (check CPUID NX feature bit) and enabled (thru boot.ini switches), OS switches to using PAE-mode
  - So it can use NX bits in the PTE/PDEs.
  - But the meaning of PAE-mode for drivers has been changed, see later ...
  - Otherwise Windows sticks to normal PDE/PTEs w/o No-Execute capability
- Appropriate switches are placed in boot.ini during install or upgrade:
  - In No-Execute capable systems, /noexecute will be placed in boot.ini
- When user-mode process tries to execute code from No-Execute Page
  - It receives STATUS\_ACCESS\_VIOLATION (0xC0000005) exception with ExceptionInformation[0] = 8.
  - Usually the STATUS\_ACCESS\_VIOLATION exception will be unhandled and result in process termination

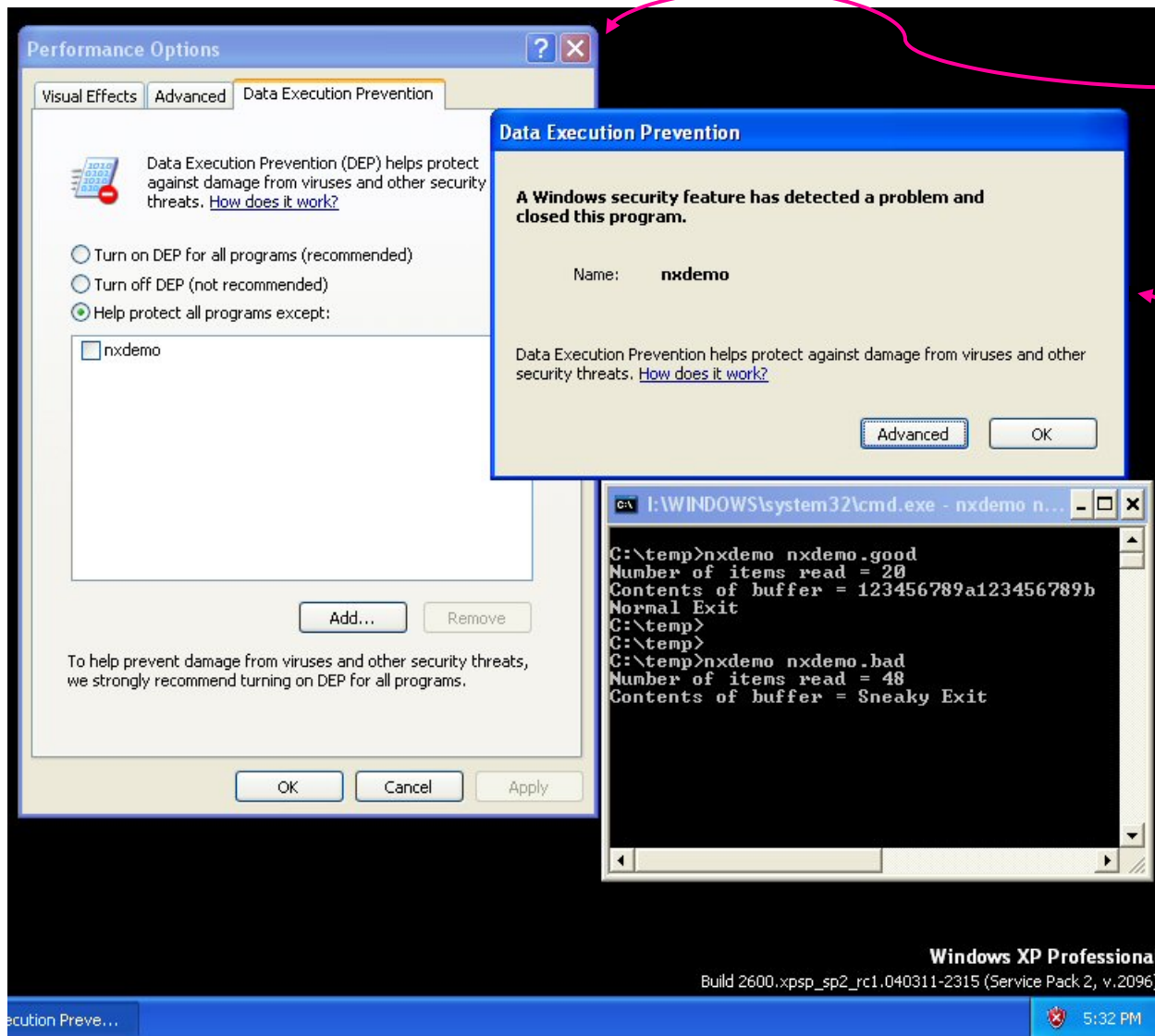
- Device drivers can't execute code from kernel stack when No-Execute is enabled. Execution protection violation in kernel-mode will result in:  
*bugcheck 0xFC: ATTEMPTED\_EXECUTE\_OF\_NOEXECUTE\_MEMORY*
- Can selectively disable No-Execute mechanism for 32-bit Apps by:
  - New application compatibility fix named “DisableNX” disables No-Execute Protection for a particular program.
  - Creates a registry string entry whose name is the Path to the application and whose value is “DisableNXShowUI”. Key is put under:  
*HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers\*
  - DisableNX can be applied to an application by using the Application Compatibility Toolkit.
- New Control Panel applet allows users to configure No-Execute globally and per-application using DisableNX mechanism above.

- /noexecute: enable No-Execute for 32-bit code (kernel and user). Implies /PAE
- /execute: disable No-Execute for 32-bit code (kernel and user)
- No switch (<blank>): disable No-Execute for 32-bit user, enable it for 32-bit kernel. /execute /noexecute = No switch behavior
- Always enabled for 64-bit kernel and 64-bit user code
  - /noexecute and /execute apply only to 32-bit user code under WoW64
- In No-Execute capable systems, /noexecute is placed in boot.ini during install/upgrade.

*“No-Execute”  
Protection is  
Enabled or  
Disabled*

boot.ini Switches:		<blank>	/execute	/noexecute
Upcoming 32-bit editions: - Windows XP SP2 - Windows Server 2003 SP1	<i>Kernel</i>	Enabled	Disabled	Enabled
	<i>User</i>	Disabled		Enabled
Upcoming 64-bit editions: - Windows Server 2003 SP1 - Windows XP 64-bit Edition  <i>for 64-bit Extended Systems</i>	<i>32-bit User</i>	Disabled		Enabled
	<i>Kernel</i>	Enabled		
	<i>64-bit User</i>	Enabled		

# What happens in 32-bit Windows® for NX Exception?



Advanced Dialog allows User to globally configure NX or disable for specific programs.

Windows handles NX exception, notifies User

Bad Input Data tricks Program to execute from User Stack, causes NX exception.

“Control Panel” →  
System →  
Advanced →  
Performance →  
Settings →  
“Data Execution Prevention”

Useful for testing:  
When App first gets an NX  
Exception, you can disable  
the App from getting NX  
again by using the Control  
Panel

This sets a per-program  
registry string entry that  
disables NX in future user  
processes that runs the App.

Value name is Application Path  
(e.g. C:\temp\nxdemo.exe)

VALUE is  
DisableNXShowUI

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers

```
I:\WINDOWS\system32\cmd.exe
C:\temp>nxdemo nxdemo.bad
Number of items read = 48
Contents of buffer = Sneaky Exit

C:\temp>
```

Windows XP Professional  
Build 2600.xpsp\_sp2\_rc1.040311-2315 (Service Pack 2, v.2096)

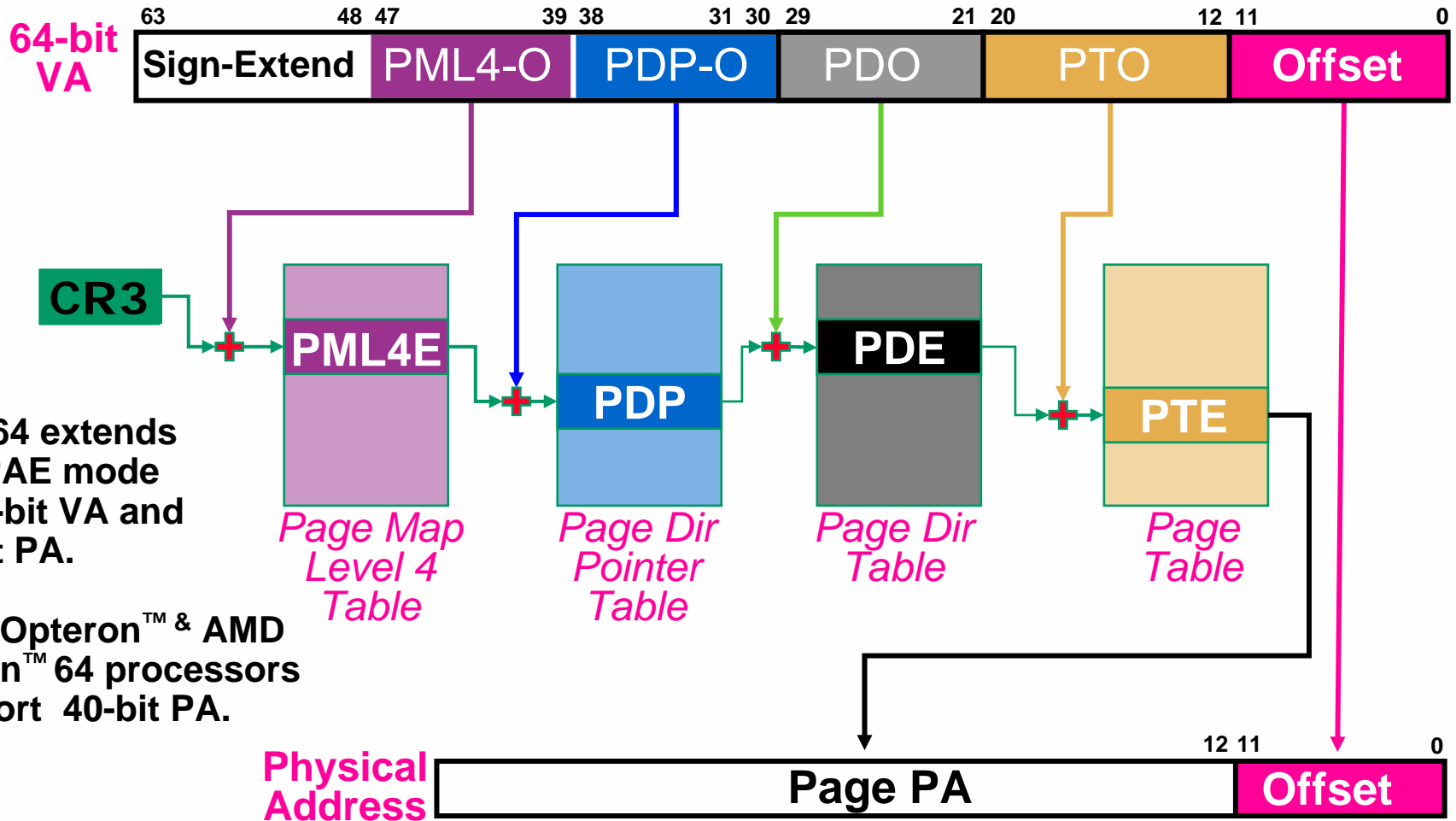
- /PAE is implied by /noexecute, but traditional /PAE behavior is a problem for most 32-bit drivers in the upcoming Windows XP:
  - Kernel uses PAE-mode and passes 64-bit physical addresses to device drivers for DMA (need 64-bits to address DRAM > 4GB).
  - Some drivers realize that the driver or the device doesn't support 64-bit physical addresses, so the driver fails to load in PAE-mode.
  - Most 32-bit devices can't access DRAM > 4GB and must rely on the kernel to "double-buffer":
    - Kernel directs DMA to a buffer below 4GB and then copies to/from the real source/target.
    - But many client drivers don't work with or lack testing of double-buffering.
    - Many drivers request an unlimited number of mapping resources (map registers) in 32-bit Editions of Windows non-PAE mode.
    - The requests succeed in 32-bit Editions of Windows non-PAE mode because the resources need not be backed by actual memory buffers.
    - In 32-bit Windows using traditional PAE-mode, each mapping resource needs real memory buffers allocated.
  - Handling requests for unlimited number of mapping resources from drivers can lead to system instability. Unless /PAE behavior is changed ...

- When Windows drivers need to do device DMA, they are required to request map registers for the DMA from the kernel.
- Map registers were invented to support devices which can not perform DMA to the entirety of the CPU's physical address space. E.g.,
  - Legacy ISA devices > 16MB; Most 32-bit PCI Bus-mastering devices > 4GB
- A map register “maps” between the CPU's source/target physical location and the address that the device uses to access this data.
  - If CPU's physical location is beyond device's addressability, the kernel will assign a memory buffer that is reachable by the device to the map register.
  - Kernel arranges for source/target's contents to be copied to/from the buffer.
  - If device can reach the entire address space in CPU's current addressing mode, kernel doesn't allocate buffers to “back-up” map registers.
  - So 32-bit devices in 32-bit Windows in non-PAE mode can get away with requesting an unlimited number of map registers.
  - This isn't true under the traditional definition of /PAE

- Solution: Kernel Limits address space in PAE-mode  $\leq 4\text{GB}$ 
  - Override the traditional definition of /PAE
  - Memory manager ignores addresses  $> 4\text{GB}$
  - *This means DRAM  $> 4\text{GB}$  can't be accessed when the solution is used.*
    - *This is a good trade-off, client desktops not yet at  $> 4\text{GB}$  DRAM ;-)*
  - Kernel presents 32-bit physical addresses to device drivers
  - HAL supports unlimited map registers because 32-bit devices can always access memory that is used by Windows.
  - Implemented for the upcoming Windows XP SP2 & Windows 2003, Standard Edition
  - Preserves device driver compatibility.
- This solution isn't implemented in 32-bit editions of Windows Server 2003, Enterprise Edition and Datacenter Edition in PAE-mode.
  - These use traditional /PAE behavior
  - Windows presents 64-bit addresses to device drivers
  - 32-bit devices get limited number of map registers and must support double-buffering
  - Windows supports much greater than 4GB of DRAM

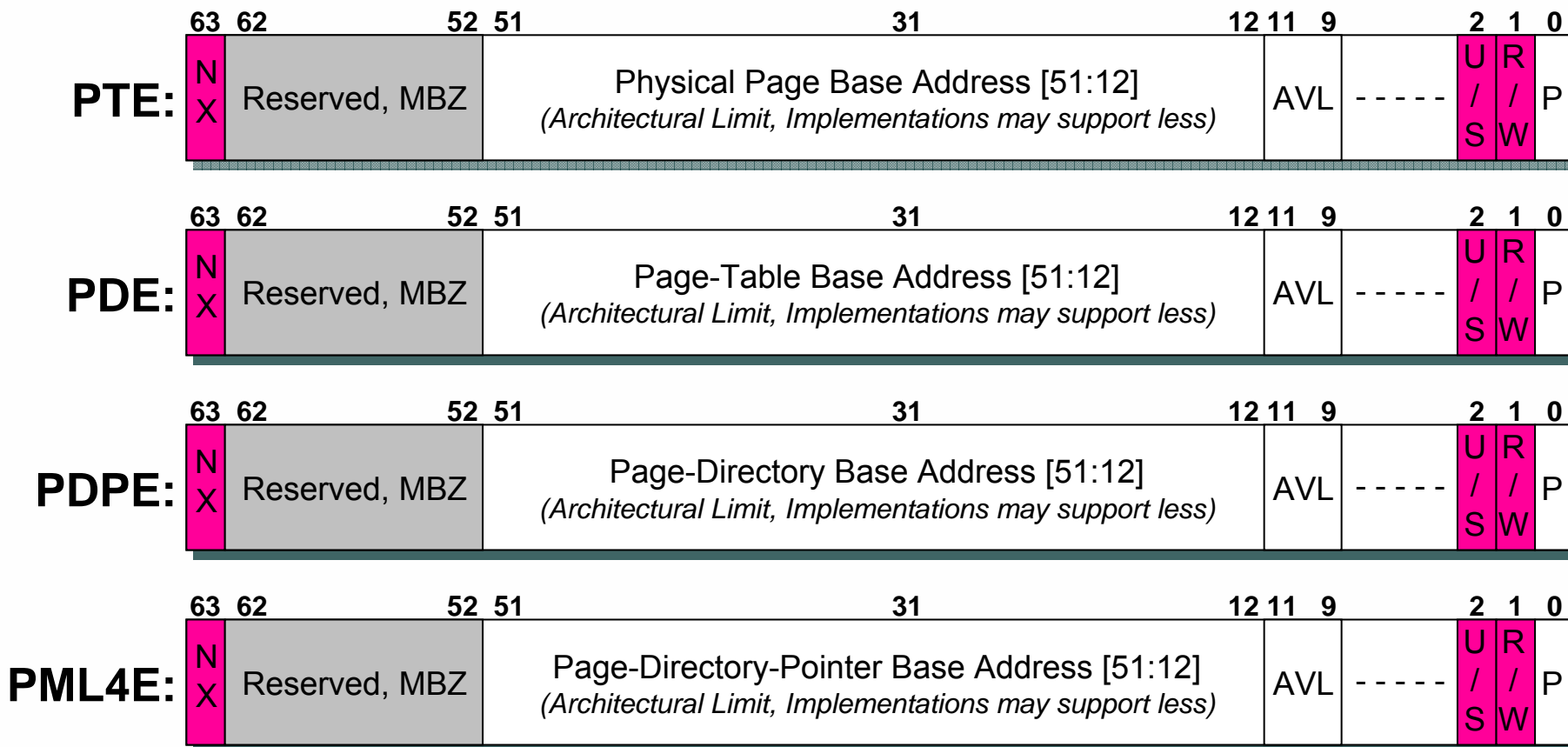
# How 64-bit Editions of Windows use AMD64 Enhanced Virus Protection

# AMD64 64-bit Long-Mode Virtual Addressing



AMD64 extends x86 PAE mode to 64-bit VA and 52-bit PA.

AMD Opteron™ & AMD Athlon™ 64 processors support 40-bit PA.



- The upcoming Windows Server 2003 SP1 and Windows XP 64-bit Edition *for 64-bit Extended Systems* support EVP.
- If No-Execute capability is present (check CPUID NX feature bit) 64-bit windows always enables it for the kernel and 64-bit user code.
  - AMD64 Long-mode Page Table entries always use PAE-format.
- BOOT.INI switches only apply to 32-bit user code (WoW64). Appropriate switches are placed in BOOT.INI during install or upgrade:
  - In No-Execute capable systems, /noexecute will also be placed
  - “/PAE” is ignored
- When user-mode process tries to execute from a No-Execute Page
  - It receives STATUS\_ACCESS\_VIOLATION (0xC0000005) exception with ExceptionInformation[0] = 8.
  - Usually the STATUS\_ACCESS\_VIOLATION exception will be unhandled and result in process termination

- Device drivers can't execute code from kernel stack, paged pool, and session pool when No-Execute Protection is enabled.
- Execution protection violation in kernel-mode will result in:  
*bugcheck 0xFC: ATTEMPTED\_EXECUTE\_OF\_NOEXECUTE\_MEMORY*
- Can selectively disable NX for 32-bit Applications through:
  - New application compatibility fix named “DisableNX” disables NX for a particular program.
  - Creates a registry string entry whose name is the Path to the application and whose value is “DisableNX” or “DisableNXShowUI”. Key is put under:  
*HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers\*
  - DisableNX can be applied to an application by using the Application Compatibility Toolkit.
- New Control Panel applet allows users to configure NX globally and per-application using DisableNX mechanism above.

# **What Applications and Drivers should do to work with AMD64 Enhanced Virus Protection**

- Apps which perform dynamic code generation (e.g., JIT'd code) or execute code from the default process stack or heap are affected.
- If your App requires “executable” memory, allocate it using the VirtualAlloc\*() API:
  - *Heap allocations using the malloc() and HeapAlloc() functions are non-executable!*
- Explicitly mark the memory “executable” by using one of these memory protection flags to VirtualAlloc\*() or VirtualProtect\*()
  - PAGE\_EXECUTE
  - PAGE\_EXECUTE\_READ
  - PAGE\_EXECUTE\_READWRITE
  - PAGE\_EXECUTE\_WRITECOPY
- Failure to do so means your App will receive an exception when attempting to execute code from those locations.
  - Exception status code: STATUS\_ACCESS\_VIOLATION (0xC0000005) with ExceptionInformation[0] = 8.

- Don't rely on No-Execute Protection as the only means of security for your App !!
  - Procedure return interception is still possible with No-Execute protection enabled (but the risk for real harm is lower)
- Reduce the risk that buffer overflow can hijack your program:
  - Use recommended programming practices in checking input data size against buffer length
  - If possible, after allocating and writing code to executable memory, block further writes to the memory using the VirtualProtect\*() API.
  - Make your executable regions as small as possible thus reducing the “attack” surface through which a virus could inject code
  - A buffer overflow in a non-executable region (e.g., stack) will grow toward higher-addressed memory.
    - Therefore, if possible, locate executable regions at a lower address than non-executable regions.

- Test your driver using both /PAE and /noexecute under the upcoming Windows XP SP2 and Windows Server 2003, Standard Edition SP1.
- It is unlikely your driver is doing dynamic code generation, but if it is, fix it to stop doing that in kernel mode.
- If your device is 32-bits and your driver can't handle double-buffering or 64-bit physical addresses, don't fail the load on detecting PAE:
  - Either fix the driver to deal with double-buffering, or
  - Use `RtlVerifyVersionInfo()` to check if you are running on the upcoming Windows XP SP2 or Windows Server 2003, Standard Server SP1
  - With either method, it then becomes safe to load the driver.
- Drivers should never directly modify the system page table entries (PTEs) so they shouldn't get confused by the format of the PTEs.
  - But if you are, fix your driver to stop doing that !!

# Other Operating System Support

- Implemented in v2.4.22/23 and v2.6 of the 64-bit kernel
  - However it is disabled by default
  - Implemented in SuSE v9.1 distribution and RedHat EL3 for AMD64
  - Not implemented in prior versions of 64-bit kernel
  - Not implemented in 32-bit versions of kernel
- Feedback:
  - Both SuSE and RedHat have NX bit implemented and see no issues at this point
  - RedHat uses it for ExecShield (e.g. Binaries with the proper elf flag turned on)
  - No specific driver issues as these were often worked out during the port to 64-bit