

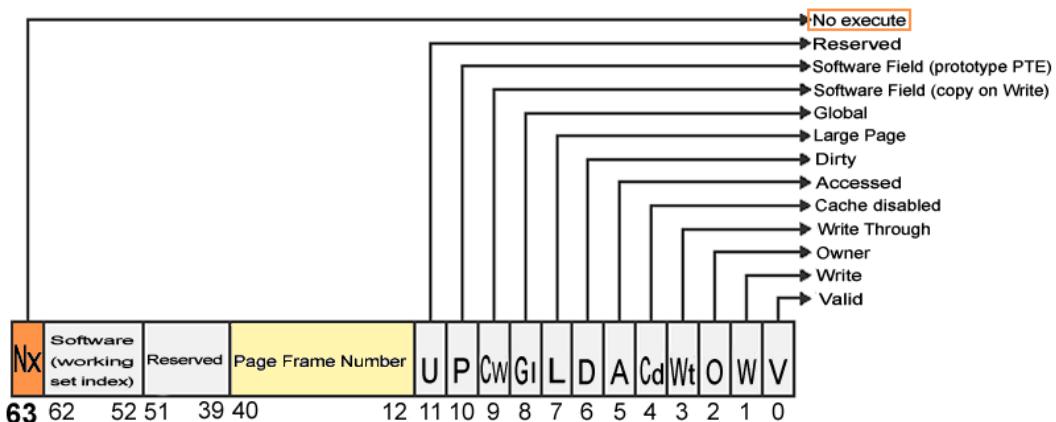
# NX bit

## A hardware-enforced BOF protection

( Argento Daniele - Boschi Patrizio - Del Basso Luca )

### What's the “NX bit”?

NX (No eXecute) bit actually refers, on x86 architectures, to the most significant bit (i.e. the 63th, or leftmost) of a 64-bit Page Table Entry. If this bit is set to **0**, then code can be executed from that particular page. If it's set to **1**, then the page is assumed to only retain data, and code execution should be prevented.



64bit PTE (Page Table Entry) for x86\_64 or x86 with PAE kernel architectures

Many older architectures (including 80286) implemented a similar feature at the segment-level, but such a control was often considered too coarse to be useful. The more practical page-level approach has also been around for various architectures (SPARC, Alpha, PowerPC), but was firstly introduced on x86 families by AMD, for use by its AMD64 line of processors. Intel subsequently decided to market the feature as the **XD bit** (eXecute Disable). Despite the names, both perform the same function and refer to the same bit.

Although the NX bit support could be (and actually is, typically as a fallback) emulated, hardware support for execution prevention is very important. All emulation strategies included in technologies like *Exec Shield* and *PaX* proved that the extra programming logic needed to create a control of the state of the NX bit introduces a measurable overhead [2], and the fastest algorithms simply give a non-guaranteed NX approximation.

Please note that regular x86 32-bit PTEs obviously don't have any 63th bit. NX is only available with the Physical Address Extension (**PAE**) page table format [3]. This is usually not a problem on operating systems that come with support for more than 4gb of addressed memory, as they actually use PAE. In fact, PAE is supported on MMUs since the Pentium Pro era, but at that time there was no NX technology, so those processors didn't implement the logic needed to use that particular 63th bit, which was simply “reserved” or “not used”.

### How does the NX bit relate to security and buffer overflow attacks?

An operating system with the ability to take advantage of the NX bit may prevent stack and heap memory areas from being executable (e.g. impossibility to have stack/heap addresses in the EIP register). This actually means that the largest portion of buffer overflow attacks would fail, since they rely on code injection in those areas. They'll still be able to overflow buffers and overwrite memory, but the subsequent execution of the malicious payload will cause an unrecoverable access violation error. The magic is that the NX approach is purely runtime: programs don't need to be recompiled to take advantage of this protection, because standard linkers and executable formats (e.g. PE for Windows) have always used execution flags, even long before the NX bit.

So we may say that NX prevents buffer overflows attacks, system wide. But there's a notable exception. Even with this hardware protection, attackers are still able to execute some code: the one that is normally executable! A buffer overflow could indeed replace the return address on the stack of the called function with the address of a pre-existing function (e.g. `system()`), and the correct portion of the stack with the arguments to this function (e.g. `"/bin/sh"`). Such an attack is usually referred to as **return-to-libc**, since in the unix-style systems the shared *libc* library is always linked to programs, providing useful calls like `system()`, and thus being the most common target. Return-to-libc can only be prevented by **stack-smashing protections**, which usually require overhead and/or compile-time measures, and **ASLR** (Address Space Layout Randomization), which is, under some circumstances, breakable with bruteforce, at least on 32-bit systems [4]. Add the fact that applications could also `mprotect()` their pages at runtime. The typical suggestion is that NX gives its best when used in conjunction with other protection schemes (being ASLR his best friend).

## Does my CPU support the NX bit?

You can check this under the DEP pane (see next section) in Windows, or if you're under Linux you can try with a `grep nx /proc/cpuinfo` command. However, concerning x86 families, any 64-bit processor supports the NX bit. This includes AMD Athlon 64, Opteron, Turion64, K8; Intel Itanium, Pentium-M (later revisions), Prescott and later iterations, Celeron D, Core Duo; Transmeta Efficeon and newest VIA processors.

If you're confused, simply use a cpu identification utility, or issue the following ASM instructions:

```
MOV EAX, 80000001h (CPUID Extended Features)
CPUID
```

Bit n. 20 (e.g. the 21th counting from right) of the EDX register should be set to 1 if NX is supported.

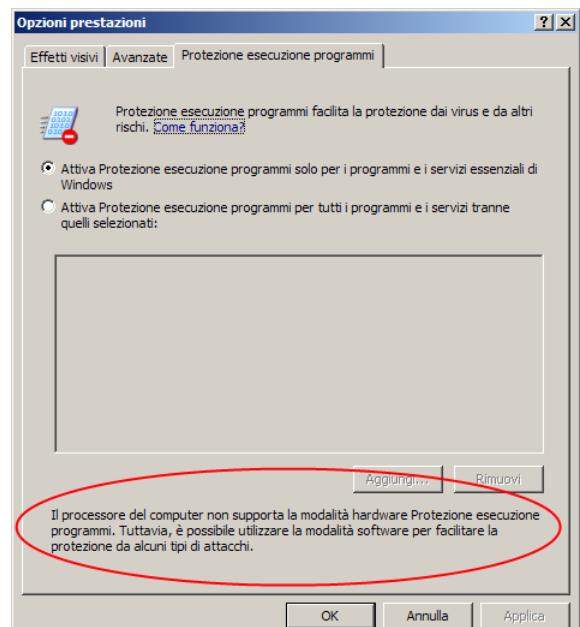
## Does my Operating System support the NX bit?

Many operating systems implement a NX policy, and some implement or have available NX emulation (which will be not covered in this document). Note that the NX bit doesn't require the host to run a 64-bit OS, assuming you're using PAE.

### Microsoft Windows

The feature is actually called DEP (Data Execution Prevention), and it's available starting from Windows XP SP2, Windows Server 2003 SP1, and Windows Vista. There are no current plans for backports. Windows XP Media Center Edition 2005 and Microsoft Windows XP Tablet PC Edition 2005 are both based on XP SP2, so they also support the feature. If your CPU doesn't support the feature, you'll get a notification stating it will use a software enforced DEP protection. This is *not* an emulation, and in fact it's completely unrelated to execution prevention, as it protects from another type of attack (*SEH overwrite*).

DEP can be enabled either for all programs (*AlwaysOn*), for all programs with some exceptions (*OptOut*), only for core components (*OptIn*), or can be disabled (*AlwaysOff*). By default, in home/pro/desktop editions, it's *OptIn*. To change this, simply edit the `C:\boot.ini` file and add the `/NoExecute:OptOut` option [5]. You can also do this using the Windows GUI, under Control Panel -> System -> Advanced -> Performance -> Settings -> Data Execution Prevention. Here you can also add exceptions, e.g. for programs that don't support NX, like old versions of the Divx Encoder.



As previously mentioned, the OS *needs* to run PAE. Microsoft documented that the functionality is automatically enabled if you specify any DEP related option in the `boot.ini` file, but this happened not to be true on some systems with a preinstalled Windows XP. To be *completely* sure that DEP is *really* enabled, you should run a software security check. If it's not, then add the “`/pae`” option in `boot.ini`, and blame Microsoft if you read this after an hour of checks and reboots.



DEP process termination.

## Linux

Linux supports NX bit since the 2.6.8 kernel (August 2004), but doesn't take advantage of it without a proper protection technology installed. The most famous technologies are *Exec Shield* and *PaX*, the latter included in the `grsecurity` set [12] and enabled by default on **Hardened Gentoo** and **Trusted Debian** distributions.

The mentioned patches, and in particular *PaX*, offer a complete least-privilege protection for memory pages, flagging data memory as non-executable, program memory as non-writable, and randomly arranging the address space. The policy to take advantage of the non-executable memory pages (PAGEEXEC) correctly uses the NX bit and, if not found, will emulate it in various ways (e.g. the overloading of the Supervisor bit), at the cost of a performance detriment. To enable it, the easiest way is to apply the `grsecurity` patchset, or to download a pre-patched kernel (e.g. *hardened-sources* on Gentoo, *cr0 kernel* [12] on Ubuntu), and recompile it with the `PAX_NOEXEC` symbol set to yes (in kernel menuconfig this is located at Security Options -> PaX -> Enable various PaX features -> Non-executable pages).

Some desktop distributions like **Ubuntu**, **openSUSE** and **Fedora Core 6** do not enable the `HIGHMEM64` option by default, which is required to gain access to the NX bit on 32-bit x86 kernels. This is due to the fact that the PAE mode causes many old processors (pre-Pentium Pro, AMD K6 and earlier, VIA C3 and similar) to fail to boot. It is however safe to enable this option if you have a recent CPU.

## Other

We only tested Windows and Linux platforms, but FreeBSD (April 6, 2007), MacOS X for Intel (10.4.4) and later are stated to support NX technology. NetBSD 2.0 has a per-page NX support for all architectures except i386 and powerpc that don't have the bit, which have a region-based protection.

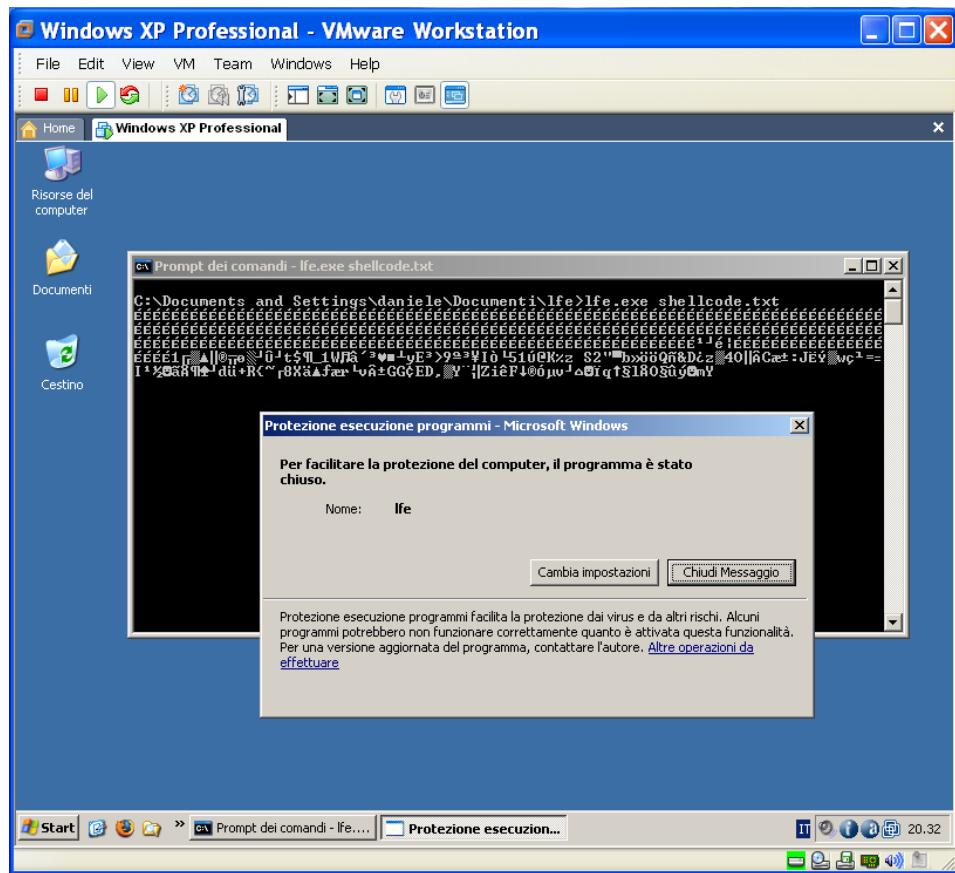
OpenBSD 3.3 (and later) offer the **W^X** (which stands for Writable XOR eXecutable) technology, recently taking advantage of the NX bit.

Solaris has supported stack execution disabling on SPARC processors since Solaris 2.6 (1997), and as of Solaris 10, NX is automatically enabled by default on x86 processors that support the bit. It's nice to hear that, stating to Wikipedia, Sun recommends that failures due to NX-like technologies should be reported as program bugs.

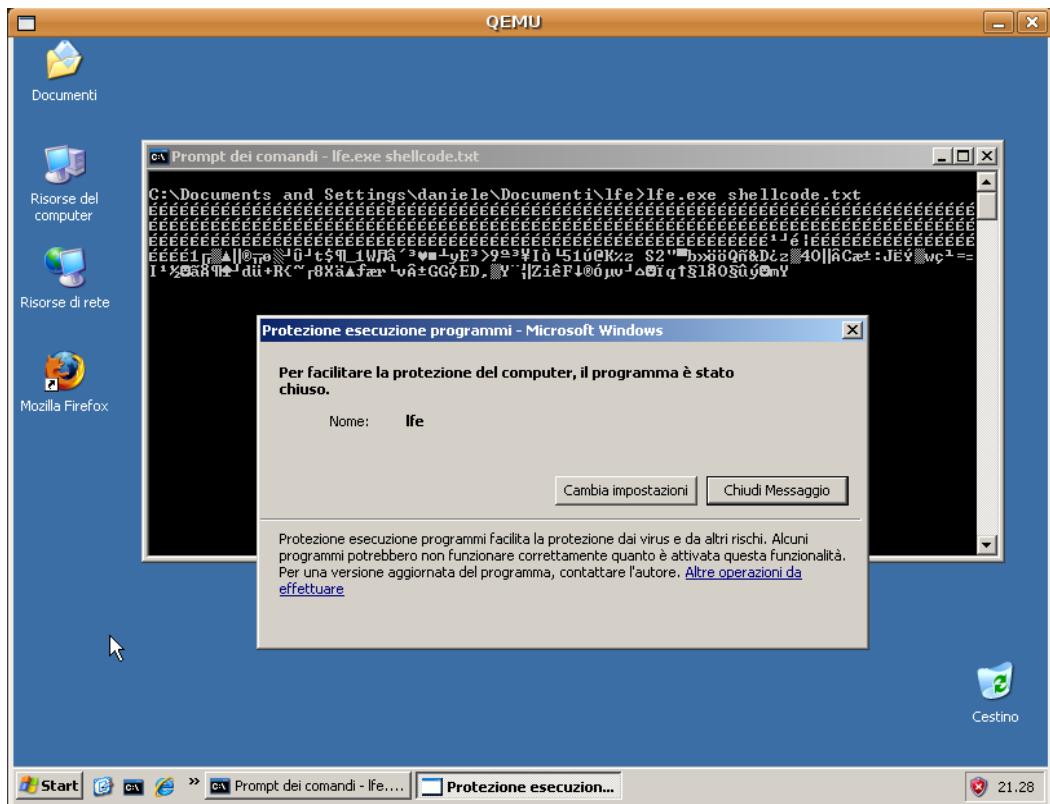
## Does my Virtual Machine support the NX bit?

**Short answer: yes.** A virtual machine (VM) is a software implementation of a machine (computer) that executes programs like a real machine. System virtual machines (sometimes called hardware virtual machines) allow multiplexing the underlying physical machine between different virtual machines, each running its own operating system.

The types of VMs we have tested are **VMWare Workstation 6**, that is a virtual machine software suite for x86 and x86-64 computers from VMware, and **QEMU**, that is a processor emulator that relies on dynamic binary translation to achieve a reasonable speed while being easy to port on new host CPU architectures, developed by Fabrice Bellard. Both the VMs have introduced the NX bit support in their implementations: VMWare Workstation from the version 5.0 and QEMU from the version 0.8.0. In more detail, we have tested the NX bit protection support capability of VMs in a Windows XP Professional (32 bit) guest on a Windows XP Professional (32 bit) host with VMWare Workstation 6, and on a Linux Ubuntu Gutsy 7.10 (32 bit) host with QEMU, both based on an AMD x86\_64 bit processor hardware.



DEP on a **Windows XP (32) / X86\_64** guest running on a **VMWARE** Windows XP (32) host AMD64 equipped.



DEP on a Windows XP (32) / x86\_64 guest running on a QEMU Linux Ubuntu (32) host AMD64 equipped

## Does the NX bit really work?

**Short answer: yes.** An NX enforced protection technology (like DEP on Windows or *Pax* on Linux) will prevent a buffer-overflow-like attack to run his payload. You can check this by yourself using one of the existing security checkers (like the *COMODO BO Tester* on Windows [6], or *paxtest* on Linux), if you don't feel comfortable with a real-life exploit on your own machine.

### Windows

If you want to check Windows DEP personally, you could try to play with the included and well documented C programs. In particular, the *lfe.c* program is a simple and very lame file reader (like *cat*, which is not lame), with an evident BOF vulnerability for >200 byte input files (just check its source and comments). Passing him an *evil* text file, from now on a "torpedo", consisting of some (hint: 220 bytes) padding, an ad-hoc return pointer (will explain this later), and a shellcode (from now on: the simple and innocuous "Windows Execute Command (cmd.exe)" shellcode from **Metasploit** [7]), you should get a new and fresh exploited prompt if you don't have DEP enabled. Otherwise, you should get an error (or nothing at all, which is a bit disturbing since Windows should always pop a security notification). The included *windows-lfe-torpedo.txt* should work for you out-of-the-box, but you can of course create your own exploit (i.e. with a [NOPs][SHELLCODE][<-RET][\0] pattern - Hint: don't even try this) [11].

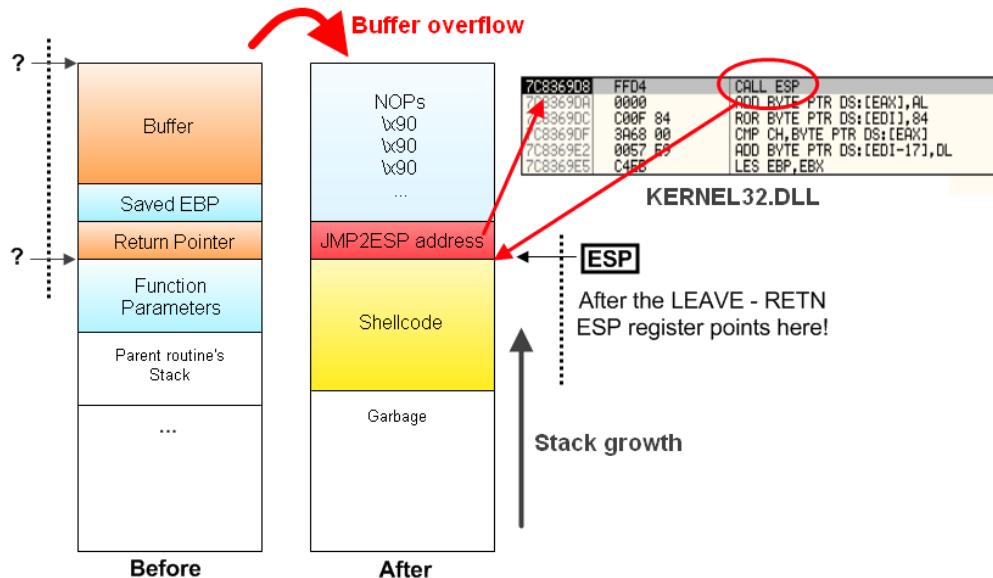
C:\>lfe windows-lfe-torpedo.txt  
C:\>Microsoft Windows XP [Versione 5.1.2600]  
(C) Copyright 1985-2001 Microsoft Corp.  
C:\>

**DEP not enabled.** Shellcode has been executed.

C:\>lfe windows-lfe-torpedo.txt  
C:\>Microsoft Windows XP [Versione 5.1.2600]  
(C) Copyright 1985-2001 Microsoft Corp.  
C:\>

**DEP enabled.** Silent crash with no shellcode execution.

If you don't want to use the included *torpedo*, and you are having troubles generating your own with an hex editor, you could try the included *torpedo-gen.c*, which will generate the classical [PADDING][RET][SHELLCODE] pattern on a file, using the before-mentioned shellcode. Simply run it with *torpedo-gen donotopenthis.txt 220 0xCAFEBAE* and you'll get malicious food for *lfe*. The CAFEBAE, you know, is the tricky part. In clean and simple theory, it's [RET], and it should be replaced with the address of the first byte of the payload. So when the function *returns* (LEAVE – RETN in assembly), the execution flow will jump to the shellcode. In dirty practice, you probably don't know that address, and even if you do, it does contain a NULL (**00**) byte, because the stack is usually located at some **0x0077BABA** like address, and **00 is a no-go on char-based overflows** (usually you can only have ONE null byte per buffer overflow attack: the terminating one). However, just after the RETN instruction, you should notice that the ESP register contains, guess what, the Stack Pointer, which is, in fact, pointing to our [SHELLCODE], as it resides in the stack just after the [RET] eaten by the RETN. So the trick is: return to a JMP ESP instruction located somewhere in the whole universe and, through it, re-return to the first [SHELLCODE] instruction! This is, again, clean (?) theory; dirty practice says that the address of a JMP ESP is not easy to find at runtime. But you can rely on the fact that libraries like *ntdll.dll* or *kernel32.dll* are always loaded, so you should be able to find there a "JMP ESP", or "CALL ESP", or "PUSH ESP .... RETN" instruction(s), and use them (hint: use *findjmp2.exe* [8], or try with the usually valid 0x7C8369D8 address, a nice JMP ESP kindly offered by Microsoft on *ntdll.dll*).



The “JMP2ESP” approach for stack execution buffer overflows

## Linux

If you’re under linux, try with the `linux-lfe-torpedo.txt` file. It uses the magic `0xFFFFE777` address (found with the included `linux_findjmp.c`, which is a *very* basic port of the `findjmp2.exe` utility for Windows) and a simple “print hello to stdout” shellcode. It should work out-of-the-box on Gentoo and Knoppix distributions.

Note that under Linux you’ll have difficulties testing the NX protection alone, since any recent kernel (>2.6.12) comes with a simple low-entropy ASLR for the stack, and sometimes the `-fstack-protector` compile option (a canary protection) of GCC is enabled by default. To avoid these problems:

- `echo 0 > /proc/sys/kernel/randomize_va_space`
- Compile with the `-fno-stack-protector` option (otherwise you’ll get “stack smashing detected” canary messages)

Once you’re sure that no other protection mechanism is interfering with your testing, and assuming you’re using PaX or grsecurity, you can use the `paxctl` utility to enable and disable the various PaX protection schemes. The options are:

- `paxctl -psm lfe` to disable the protections;
- `paxctl -PSM lfe` to enable the protections.

The screenshot shows a terminal window titled "crono@crono-ubuntu: ~/test". The user runs the command `./lfe linux-lfe-torpedo-sh.txt`. The terminal output shows a series of characters being printed, followed by a segmentation fault (SIGSEGV) indicated by the address `0000V` and the instruction `^Z100F2F`. The terminal then displays "bin/sh" and "Killed", indicating the exploit was successful but the process was terminated.

You’ll get a “Killed” message on your shell whenever you attempt to execute stack or heap.

## Are the current NX implementations safe?

Short answer: yes and no.

First, it's widely known that some programs (e.g. video encoders) sometimes, for efficiency, need to generate code on-the-fly and execute it. Obviously this has to be done in heap or stack space, and that's exactly what NX implementations try to prevent. The solution, in these circumstances, is to add an exception rule for the program, or to wait for a new version of the program (with added linking-time measures, like X/E flags in the interested memory portions). There are some other notably incompatibilities with No-Execute technology: wine and some old versions of Xorg. However you can usually add exceptions for these programs.

Second, current implementations doesn't seem to be *perfect*. It's just a matter of searching "bypassing dep" or "bypassing nx" on google to get confirmations to this. As an example, the Windows *OptOut* mode (the one with a per-process exclusion) makes the protection weak, because the disabling of NX support for a process has then to be determined at execution time. To support this, the routine *LdrpCheckNXCompatibility* was added to ntdll.dll, which in its execution makes a call to *NtSetInformationProcess* to either enable or disable the MEM\_EXECUTE\_OPTION through its parameters. One method of accomplishing an attack to *disable* DEP for a process and THEN execute the shellcode would be to use a ret-2-libc attack which at some point transfers the execution flow to the *NtSetInformationProcess* function with a well-formed frame set up on the stack. The drawback is that this method requires, on a straight approach, various 0 (NULL) bytes as part of the buffer that is used to overflow (and to build the well-formed frame).

Although we failed at the attempt to create such an exploit with a string based buffer overflow, with a debugger [9] it's not so hard to simulate the *ret-2-libc* attack by manually entering ad-hoc parameters on the stack and therefore chaining some jumps that actually disable DEP for the process [10].

## Conclusions

If you have Windows, enable DEP, and get some free protection, but don't *rely* on it, and remember to add exception rules for programs that begin to crash or complain about it. Also remember that Windows XP doesn't have ASLR, thus you're not protected against *all* kinds of buffer overflows. If you're using Linux... well, you should read about the PaX and grsecurity kernel patches, and decide by yourself if you really need them, as they come with other protection schemes and are not easy to install for the home user.

	XP 32bit	Vista 32bit	XP 32bit (QEMU)	XP 32bit (VMware)	Linux 32bit	Linux 64bit
x86	-	ASLR	DEP <i>x86-64</i> <i>virtual.</i>	-	PaX <i>Software</i> <i>emulation</i>	
x86-64	DEP + ASLR	DEP	DEP	DEP	PaX	PaX

A summary of runtime BOF protections for a selection of representative OSes

## Links & Bibliography

---

- [1] NX Bit on Wikipedia [http://en.wikipedia.org/wiki/NX\\_bit](http://en.wikipedia.org/wiki/NX_bit)
- [2] Pax benchmark <http://www.pjvenda.org/linux/doc/pax-performance/>
- [3] How PAE X86 works <http://technet2.microsoft.com/windowsserver/en/library/00284c8d-7a42-40f2-8a01-8de61dccc8c91033.mspx?mfr=true>
- [4] Shacham et al. "On the Effectiveness of Address-Space Randomization"  
<http://www.stanford.edu/~blp/papers/asrandom.pdf>
- [5] Boot INI Options Reference <http://technet.microsoft.com/en-us/sysinternals/bb963892.aspx>
- [6] Comodo BO Tester [http://download.comodo.com/cpf/download/setup/utility/Setup\\_BOTester\\_x32.exe](http://download.comodo.com/cpf/download/setup/utility/Setup_BOTester_x32.exe)
- [7] Metasploit <http://www.metasploit.org/>
- [8] Findjmp2 <http://packetstormsecurity.org/>
- [9] OllyDbg debugger <http://www.ollydbg.de/>
- [10] Skape et al. "Bypassing Windows Hardware-enforced Data Execution Prevention"  
<http://uninformed.org/?v=2&a=4&t=sumry>
- [11] Aleph One. "Smashing the stack for fun and profit". Phrack Magazine, 49(14), Nov. 1996.  
<http://www.phrack.org/phrack/49/P49-14>
- [12] Grsecurity patchset <http://www.grsecurity.net/>
- [13] cr0 security kernel for Ubuntu <http://kernelsec.cr0.org/>