

ActionOutLine, small code...usefull utility...and (believe or not) well protected!

*First I want to say at the author/s of this little and simple proggy...THANKS!... for two reasons, one because the main idea of ActionOutLine is so simple but so useful (at least for me) and to be honest today is very difficult to see really useful programs. Second because this little jewel is defended very well against the novice and medium crackers attacks...here change some bytes or skip some code or reset some flags is not enough...
...welcome in the real world of reversing!*

(please forget ...my crappy english ☺)

(Modified slightly by amante4)

Well, someone probably in this moment think... 'Hei ..Mav...this is not a software protection system...It's a normal program...' he! he! true! But don't worry we have enough here for a real good fun...

I've met this proggy after that I've read a message posted on the Italian Cracking Forum (Ahh thanks Xasx for the board!)...a guy was desperate because a crack just downloaded didn't remove the limit of 7 records...but only the 30 days of evaluation. After some posting the guys was very depressed... why?? What had this proggy of so special?? ...well I decided to download it to see why there wasn't a full crack on the net...here what I've found...

Target:

ActionOutLine 1.6 <http://gpsoft.hypermart.net>

Tools:

- Sice 4.x
- Hex editor (choose what you want)
- Brain (...if you are a Micro\$oft coder...well... search this tool with <http://www.altavista.com/> ☺)

The first protection in the code is the check on 30 days of evaluation....This is not a problem and It's not interesting for us....the second protection is the limit of 7 records (or nodes, or items call it how you want...in this tut I'll refer to it as nodes)...more interesting and more important...but the main thing here is that these 2 different checks are not related with the state of the proggy...registered or unregistered....for the simple reason that the program works only in trail mode and there's no registration option! This means that the proggy has no code to work properly in full mode. We see now in which way I 've arrived at this conclusion and in which way we can fix the problem....

OK run the target....the first window that we can see is the nag for 30 days of evaluation....not interesting....go on...now in the main proggy...we can create a new tree of nodes....

Under the 'root' we create a node...then under this node ...we create 7 nodes....better explained with the picture below:



At this point we try to insert another item the 'eight' ...the nag pops up ...to remind us that we have reached the limit...

Ok...the common and easy thing to do is find the call at this nagscreen...

```
:0046FBEA    mov eax,ebx
             call 00454440
             cmp eax,07
             jl 0046FC07
             lea edx, [ebp-04]
             mov eax, [00475D8C]
             call 004049C8
             mov eax, [ebp-04]
             call 004682F          ;Here we have the Nagscreen
```

Well...to be honest ...with only these few lines of code...a Micro\$oft coder could understand that the cmp eax,07 checks the number of nodes.....very easy...and very wrong!!!!

OK...why....??? We need to keep our ideas cleared...because we need some reversing skills...go ahead with a right path...

The simple check seen before is correct....what the program does is check if the number of nodes inserted is less than 7...well if not...nagscreen!

The fast thing to do is reverse the jl to skip the nag's call....at this point the program....he heshows a second different nagscreen...or better this is a classic messagebox. It reminds us that only 7 nodes are permitted....uhmm...quite enough to understand that there's a second check...

We step the code after the reversed jump to find where the messagebox is shown.

```
:0046BB9A    mov eax, esi
             call 0045444C
             mov esi, eax
             jmp 0046BBC6
             ... ..
```

We need to step in the call 0045444C to see the real call at the messagebox, here the code:

```
:004542F1    test eax, eax
             jz 004542FC
             mov edx, ebx
             call 004543A0          ;show the messagebox
             pop esi
             pop ebx
             ret
```

If we skip the call ...the job is done...no more nags or messages...and the program insert in the tree our new node. Good ! we can call it 'eight' and the program seems cracked...but if we stand up we receive a big kick in the ass....yeah the proggy has joked with us!....

OK...now we need only to save all and close the program....then reopen it and load the file saved...to see if our cracked code works fine....he he!

Big surprise! The proggy seems that discards the data over the 7 permitted...two can be the situations here...one, the proggy cracked saves all data (over the seventh too) but when it loads the file, it checks for the presence of non-permitted field (over the seventh) and loads correctly only seven of it...

Second, the proggy cracked...when we insert an eight node, it shows the information only in the tree, but it doesn't save data over the seventh insert ed.

Well for us...the more easy situation is the first...and from this we start to see if the problem is in the loading data code....

First we need to check the saved file to see if the information are stored or not...

Open the our saved file with an hex editor...better Ultraedit ...and we see what we can see ☺

```

00000000  5B47 505D 5B41 4F5D 5B31 2E30 5D5B 3031  [GP][AO][1.0][01
00000010  5D08 0000 0000 0000 0004 0000 0052 6F6F  ].....Roo
00000020  7400 0000 0007 0000 0003 0000 006F 6E65  t.....one
00000030  0000 0000 0000 0000 0300 0000 7477 6F00  .....two.
00000040  0000 0000 0000 0005 0000 0074 6872 6565  .....three
00000050  0000 0000 0000 0000 0400 0000 666F 7572  .....four
00000060  0000 0000 0000 0000 0400 0000 6669 7665  .....five
00000070  0000 0000 0000 0000 0300 0000 7369 7800  .....six.
00000080  0000 0000 0000 0005 0000 0073 6576 656E  .....seven
00000090  0000 0000 0000 0000 0000 0000 0000 0000  .....

```

How we can see our god Star this night is not passed on our head! ...and the proggy doesn't save the eight node...we are in the second situation described before. We need to study the code that performs the saving operation...Ok...no fear! And go!

We open our saved file and we insert again the node 'eight' now we know in which way ☺

After we set a bpx on DialogBoxIndirectParam api function.

This 16bit function is used (for compatibility with 16bit apps) to create a savedialog...another classic example that Win98 is the same of Win95 that is the same of WfW3.11 advanced mode...same shit!

But this is another story.

Anyway... Sice pops upwe need to renter in the proggy code...

OK...now in code we know that the proggy at this point has retrieve the name of the file chosen by us to save all. And now it create the file and start to write in it! Yeah...write in it...we set a bpx on

WriteFile api function....and...

Sice breaks here...

```

:00407077  push 0                    5 parameter
                lea eax, [esp+04]  offset Dword with num bytes written
                push eax          4 parametre
                push edi          3 parameter num bytes to write
                push esi          2 parameter offset buffer with bytes to write
                push ebx          1 parameter file handle
                call kernel32!writefile  write!

```

How we can see if we leave the proggy run...this is the only piece of code for writing the dat as in the file...good for us!

We put a bpx on the 2 parameter (push)...to see the content of the buffer that stores the bytes to write. Everytime the code use writefile to save on disk we can see what byte it saves...and understanding the format used.....I know, I know!...someone at this point can ask...'why to understand the data format used to save our data ???' ...because we need a clear idea about where the proggy store all our data just before to save on disk...I mean array, variables, pointers etc.

To make the thing easy for all readers....I don't explain every lines of code...this is quite easy to do in Sice and it doesn't take a lot time...but I'll explain directly how the program save our data in a file...

Ok...fireup Sice and see what is happened....

I've put all information collected at every break in a table to keep our mind and the tut quite readable and understandable ☺.

The table shows, for each call at writefile, the number of bytes to write, the bytes to write and the addresses where the proggy retrieve these 2 info. Well...we can give a look....

Num of call WriteFile	Address with Num bytes to write	Num bytes to write	Address with Bytes to write	Bytes to write
1		17		[GP][AO][1.0][01]
2	006FFAD8	8	006FFB30	00000008
3	006FFA94	4	006FFAD0	0004
4	006FFA9C	4	00C34664	Root
5	006FFA94	4	006FFAD0	0000
6	006FFAA4	4	006FFAE0	0007
7	006FFA60	4	006FFA9C	0003
8	006FFA68	3	00C21120	One
9	006FFA60	4	006FFA9C	0000
10	006FFA70	4	006FFAAC	0000
11	006FFA60	4	006FFA9C	0003
12	006FFA68	3	00C34AFC	Two
13	006FFA60	4	006FFA9C	0000
14	006FFA70	4	006FFAAC	0000
15	006FFA60	4	006FFA9C	0005
16	006FFA68	5	00C22C48	Three
17	006FFA60	4	006FFA9C	0000
18	006FFA70	4	006FFAAC	0000
19	006FFA60	4	006FFA9C	0004
20	006FFA68	4	00C34B7C	Four
...

I've cut the table...but we can see here a lot of useful information. First we have a clear idea about in which way the proggy organizes the data in a file. In fact at the first writefile call it writes 17 bytes. It seems like a little header with name of the proggy and version and others unknown info.

After it writes 8 bytes, value 00000008 this is the total number of nodes (included the first 'Root').

Then it writes 4 bytes, value 0004, this is the number of characters of the node's name that follow.

After it writes the name of the node (Root). At call 5 the proggy write 4 bytes, 0000 , with the info that we have here is difficult say what these bytes means....but when I wrote this table...I went more in depth, just for my curiosity...to find the answer at this question mark ☺. The answer is quite simple...for each node that we insert in the tree we have the option to insert some text in the right side of the main windows....and ya! These 4 bytes are the number of characters of the text inserted for the specific node. In our case we haven't inserted text for any node...if we had inserted some text now after these 4 bytes...the proggy wrote the entire text...

Ok we can go on, the 6th call writes the number of subnodes for the first node (Root), value 7. Then writes the number of characters of the name for the first subnode, 3.

After it writes the name, 'one'. At the 9th call restart the same cycle above...for the current node, 0000 (num of chars for the text associated), 0000 (num of subnodes for this subnode).

At the call 11 restart the cycle for the 2nd subnode (two)... and so on....not difficult.

An important thing reported in our table is the address of each information written to...we see that all addresses comes from memory allocated at runtime, this is important because means that the proggy doesn't use arrays fixed in size to store a predefined number of information...

OK...enough...

Now we make a step behind...at the code seen in the start of tut...more exactly here:

```
:0046FBEA    mov eax,ebx
              call 00454440          Retrieve the number of nodes created
              cmp eax,07             Are the number of nodes created less of 7 ??
              jl 0046FC07            yes jump, no nag
              lea edx, [ebp-04]
              mov eax, [00475D8C]
              call 004049C8
              mov eax, [ebp-04]
              call 004682F           ;Here we have the Nagscreen
```

We need to step in the call 00454440 to see where the proggy store the number of nodes created...

Here that we find inside this call:

```
:00454440    mov eax, [eax+20]          eax = num of nodes created
              ret
```

eax = 00C349B4

eax+20 =[00C349D4] = 00000007

Ok...a part that this is probably one of the faster call of all program (2 lines!!!)© ... the address where is stored the number of nodes is dynamically allocated...and is obviously different from pc to pc and it could be different at every run. What is really interesting here is not this address but the way used to refer at this info.

[eax+20]

This is an array and at the index 20h the program store the current number of nodes. This array starts at the address 00C349B4, better if we give it a look:

:00C349B4

78	41	45	00	84	A3	C6	00	CC	6D	C2	00	04	6E	C2	00
A4	42	C2	00	DC	42	C2	00	14	43	C2	00	C0	56	C2	00
07	00	00	00	D8	18	C7	00	00	00	00	00	00	00	00	00

Humm...look nice! ☺ A part the first dword (78414500) that seems a marker...the others dword are simple pointer, and again in allocated memory. Take the first pointer (00C6A384) and go to see what we find:

:00C6A384

78	41	45	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	20	11	C2	00	00	00	00	00	B4	49	C3	00

Humm....look more nice too! OK...first dword we assume that it's a marker...after we have a lot of zero bytes...then a pointer again in allocated memory (00C21120)...but wait...we have already seen this address...meanwhile you think where we have seen it...I take the classic moment for the toilet...sorry but I drink a lot when I reverse code ☺

...10 min later (it was a lot ☺)

...OK...we have seen this pointer in the first table when we have studied the saving code...exactly at the writefile call number 8...if we check what we have at this address we find the string 'one' (the name of our first node) like we've already seen in the first table...do you remember ??

Good!...but now we go a step forward...

If we look at the 2 arrays above...we see something of really interesting and important...what??... simple the 2 arrays are identical !!!! ...I know someone can say...' hei Mav are you drunk ?? ...the only bytes that are the same in both arrays are the marker at the start (78414500).....'

He! he!... well... this is true...fucked true! ...and it's been exactly the marker that has turned on the light in my brain ☺.

Both arrays are different only in the content (obviously), but are the same in the format...the first array is been created for the first node (Root), the second array is been created for the second node (one)...

The reason for all that zero bytes in the second array...are simple, our second node (one) has no subnode!!! And for this reason there's no pointer in the array at other arrays!! But zero bytes!!

OK...make some order in our brain:

Everytime we create a new node in the tree, the program create an array like the above array...same size and same format...here:

78	41	45	00	84	A3	C6	00	CC	6D	C2	00	04	6E	C2	00
A4	42	C2	00	DC	42	C2	00	14	43	C2	00	C0	56	C2	00
07	00	00	00	D8	18	C7	00	00	00	00	00	00	00	00	00
00	00	00	00												

- 1 dword, seems to be a marker.
- 7 dword , pointers at each subnodes for this node (max 7 subnodes for each node!)
- 1 dword, this is the number of subnodes for this node (the limit is 7)
- 1 dword, pointer at the name of this node (ascii string)
- 1 dword, pointer at the text inserted for the node
- 1 dword, pointer at the parent array. If this array is for a subnode, this pointer is the address of parent array (node).
- 1 dword, this dword is the flag that we can set for each node, there are different colour (read the manual) ex. If we set a red flag for the node, this dword will be set at 02 00 00 00

Well...now we have some information in more...how we can see the size of these arrays is 48 bytes, and the space to store pointer at the subnodes is fixed at 28 bytes (7 subnode * 4 bytes).

We see that if we create an eighth node and we force the code to write the address in the array, it overwrites the dword that store the current number of the nodes (red rectangle). This situation obviously generates an exception in the code.

The first thing to do...is...find the part of the code that alloc the mem for each array, and see if we can increase the size of these arrays without creating problems. After that we need to take care of the information stored in the array after the dword of the current number of nodes...but wait one thing at a time...

To study the code that allocs mem for the array, better if we create a new node under ,for ex., the node 'one'...in this way we don't need to bypass the checks for the limit of 7 nodes.

The proggy does n't use VirtualAlloc to allocate memory, forget bpx on apis now...time wasted...believe me!

The point to start is where the proggy checks the number of nodes already created....it compares the number with 7 ...do you remember the code at the start of the tut ???

```

:0046FBEA    mov eax,ebx
             call 00454440
             cmp eax,07
             jl 0046FC07
             lea edx, [ebp-04]
             mov eax, [00475D8C]
             call 004049C8
             mov eax, [ebp-04]
             call 004682F

:0046FC07    mov eax, ebx
             pop ebx
             pop ecx
             pop ebp
             ret

:0046BB8C    test al, al
             jz 0046BC16
             cmp dword ptr [ebp-04], -01
             jnz 0046BBA5
             mov eax, esi

             Call 0045444C

:0045444C    push ebp
             mov ebp,esp
             push ecx
             push ebx
             mov ebx, eax
             mov dl, 1
             mov eax, [00454138]
             call 00454258

:00454258    push ebx
             push esi
             test dl, dl
             jz 00454266
             add esp, -10
             call 00402FD8

:00402FD8    push edx
             push ecx
             push ebx
             call [eax-0C]

:00402CF8    push eax
             mov eax, [eax-1C]

```

Retrieve the number of nodes created
 Are the number of nodes created less of 7 ??
 yes jump, no nag

;Here we have the Nagscreen

esi = pointer at the array for the node in which
 we're creating the new node.

eax = 00454178 marker ?? ☺

eax = 34h = 52 bytes = size of the array

```

      call 00402690
:00402690  test eax, eax
      jz 0040269E
      call [0047501C]

```

```

:004020D0  ...☺...☺...☺...OK...break one moment! We have stepped in every call that we have
encountered....and we are now really in deep in the code...thus fireup the torches
and pickup a helmet...brrr...it's very dark here!☺
All code seen is very easy to understand, two things I want remark...the fucked
marker is nothing else that an address...well...it was easy guess it, it points inside the
data section of the exe...could be interesting try to understand why it's put in every
array...but better if I keep my curiosity in chain...
The second important thing is the number of bytes to allocate....finally we know
where the code store this number...good for us....but give a quick look ahead...

```

```

:004020D0  push ebp
      mov ebp, esp
      push edx
      push esi
      push edi
      mov ebx, eax
      cmp byte ptr [00477410]
      jnz 004020F7
      call 00401A14
      test al, al
      jnz 004020F7
      xor eax, eax
      mov [ebp-04], eax
      jmp 0040224B
      xor ecx, ecx
      push ebp
      push 00402244
      push dword ptr fs:[ecx]
      mov fs:[ecx], esp
      cmp byte ptr [00477045], 00
      jz 00402118
      call kernel32!EnterCriticalSection
      add ebx, 7
      and ebx, -04
      cmp ebx, 0C
      jge 00402128
      mov ebx, 0000000C
      cmp eax, 00001000
      ... ..
      mov eax, ebx
      call 00401FDC
      mov [ebp-4], eax
      xor eax, eax
      pop edx
      pop ecx

```

Ebx = Num of byte to allocate.
Here the code align the size

Keep the code safe from non-reentrant problem. No more than one thread can execute this piece of code! Sure...Windows goes in crash enough for itself ☺


```

pop ecx
mov fs:[eax], edx
push 0040224B
cmp byte ptr [00477045]
jz 00402243
push 004741B
call kernel32!LeaveCriticalSection
ret

```

Alloc the number of bytes. It
returns a pointer.

Ok...what is this hell ??...this routine allocates a block of memory for the treeview object, this code is part of the treeview object and it's not been written by the coder of this proggy. Here we don't touch a byte for the simple reason that through this routine pass others requests of memory not only the memory request for each node's array.

But this is not a problem because we know exactly where the number of byte to allocate is stored...it's to update it ☺.

Well...one thing again for the code above...after the call 00401FDC (green rectangle) in eax we have the address of our the array...and proggy return from every call happy. Just after some ret from this routine, the code sets up the array:

```

:00402D68    push ebx
             push esi
             push edi
             mov ebx, eax
             mov edi, edx
             stosd                store the 1 dword (78 41 45 00) in the array
             mov ecx, [ebx-1C]    ecx = array size in bytes
             xor eax, eax
             push ecx
             shr ecx, 2
             dec ecx              ecx = array size in dwords - 1 dword
             repz stosd           clean (0) the entire array

```

Enough to try the first experiment!...we know that the number of byte of each array is stored at 0045415C, we change the value in this way:
we want 250 nodes for each node

$250 * 4 = 1000 \text{ bytes} + 24 \text{ bytes for others pointers} = 1024 \text{ bytes} = 400\text{h}$ new number of bytes

OK...we can set a bpm on the location where the program store the original value, then we restart the proggy...when sice breaks, we replace the original value with 400h...disable the bpm...and we see what happen...

What we have done is only increase the size of these arrays, but the limit is still there...no more than 7 pointer can be stored in these arrays or the code generates an exception...

Good...the proggy starts ...and works fine...we can insert new nodes...and all is regular...

Now that we are sure that the code is happy to allocate these bytes in more...we can fix the problem to reorganize the data stored in these arrays...why ??? simple!...do you remember the format used to store the data within the array ???...from the 2nd to 8th dword the code store the pointers at the 7 subarrays...and at 9th dword store the number of these subarrays!!! If we force the code to insert a new pointer (8th) this overwrites the number of arrays ...and so on...

We have two way to choose from...one patch the code to store the pointers over the 7thafter the 5 dwords at the end of the original array....

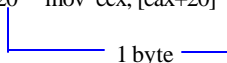
78	41	45	00	84	A3	C6	00	CC	6D	C2	00	04	6E	C2	00
A4	42	C2	00	DC	42	C2	00	14	43	C2	00	C0	56	C2	00
07	00	00	00	00	D8	18	C7	00	00	00	00	00	00	00	00
00	00	00	00
...
...

or move the last 5 dwords at the end of our new arrays of 400h bytes...like this:

78	41	45	00	84	A3	C6	00	CC	6D	C2	00	04	6E	C2	00
A4	42	C2	00	DC	42	C2	00	14	43	C2	00	C0	56	C2	00
...
...
07	00	00	00	00	D8	18	C7	00	00	00	00	00	00	00	00
00	00	00	00												

250 pointers (nodes) * 4 dword

The second solution seems the more fast, because how we can see the program use an index to write/read in the array. The form is [reg+xx] where xx is the index. By changing the index for the 5 dwords we can force the code to write/read values at different position in the array. But if look better we see that the number of bytes of this kind of instruction is 3 byte:

8B 48 20 mov ecx, [eax+20]


The index 20h is a byte, because the array is not bigger than FFh, if we move at the end these 5 values in our new array of 400h bytes we need a 2 bytes index....thus no more 3 bytes but 4 ☹. The first solution is the best (in my personal opinion...and because I'm writing the tut ... we choose this ☺). This solution seems more complex, but it isn't so! What we need to do is find where the code write/read the pointers in the array and insert a jmp at our new simple routine that check if number of the current nodes is 7 or great , if so it write/read at the correct position...
 Ok lets go to find where the proggie read/write pointers...
 We can set a simple bpm on an empty pointer in the array and after insert a new node for this array...
 Sice breaks here:

:004543A0	mov ecx, [eax+20]	ecx = num of nodes created
	cmp ecx, 7	
	j1 004543AE	if num < 7 jump, there's space in array
	call 00454300	nagscreen '...only 7 nodes...'
	ret	
:004543AE	mov [ecx*4+eax+04], edx	edx = pointer at the new array to store
	inc dword ptr [eax+20]	inc the number of nodes in the array
	ret	

Here the code uses the current number of nodes to calculate the position where to store the new pointer in the array, then it increments the number of nodes.
 Here we can simple replace the call 00454300 and ret instruction with :

add ecx, 5

Why ???...Oh shit! We are at page 10...and someone ask ...'why???'Are there Micro\$oft coders here ??? ☺ Anyway...go on...Sice breaks a second time here:

:0045437E	mov eax, eax	☺ ??????!!!!?????☺
	push esi	
	mov esi, [eax+20]	esi = number of nodes
	dec esi	**
	test esi, esi	**
	jl 0045439A	**
	inc esi	**
	xor ecx, ecx	ecx = 0
:0045438C	cmp edx, [ecx*4+eax+04]	compare edx (pointer at the new node's array just created) with each pointers stored in the array
		when it finds it...
	jnz 00454396	...in ecx = number of the pointer (0-6)
	mov eax, ecx	
	pop esi	
	ret	
:00454396	inc ecx	
	dec esi	
	jnz 0045438C	
:0045439A	or eax, -1	**
	pop esi	
	ret	

This routine return the position (0-6 for 7 pointers) where is stored the new pointers just inserted at the new node. This same routine is used when we delete a node too...this is the reason of the 5 lines that I've signed (**)...these lines check if there're subnodes for this node before deleting it...if there aren't the routine simple return with -1 in eax...meanwhile if there're subnodes it return with the position of the first zero pointer. About the first line of this routine...well...doesn't matter... we have 2Gb of virtual addresses...all crap is welcome! ☺

Ok...how patch this...humm...simple we insert a jump at new code at the end of the section...where we rewrite this piece of code patched:

:xxxxxxx	mov eax, eax	☺ Yeah!!!!☺
	push esi	
	mov esi, [eax+20]	esi = number of nodes
	dec esi	**
	test esi, esi	**
	jl xxxxxxx4	**
	inc esi	**
	xor ecx, ecx	ecx = 0
:xxxxxxx3	cmp edx, [ecx*4+eax+04]	compare edx (pointer at the new node's array just created) with each pointers stored in the array
		when it finds it...
	jnz xxxxxxx4	
	cmp ecx, 0C	
	jl xxxxxxx1	
	sub ecx, 5	
:xxxxxxx1	mov eax, ecx	...in ecx = number of the pointer (0 to current -1)
	pop esi	
	ret	
:xxxxxxx4	inc ecx	
	cmp ecx, 7	
	jne xxxxxxx2	
	add ecx, 5	
:xxxxxxx2	dec esi	
	jnz xxxxxx3	
:xxxxxxx4	or eax, -1	**
	pop esi	

```
ret
nop
```

...we have inserted new code (blue) in two different position of this routine, but see why...

- When ecx is equal to 7 means that we have an 8th pointer in our array, and to permit at the code to compare the correct pointer of this 8th (and eventually others after the 8th) we need to fix the index to skip the 5 dwords (do you remember ??).
- The reason of this 3 lines of code that I've inserted is simple, how we can see ecx is used to build an index within the array to find the pointer to compare, but not only...each pointer in the array has a specific position from 0 to 6 (for the first 7 pointer)...this means that if we insert a 8th node the 8th pointer needs to be associated at the position 7...for a 9th node the 9th pointer needs to be associated at the position 8...and so on. But with our fixup at the index in ecx, we have altered the normal position associated at the pointers above the 7ththese 3 lines fix this problem and at the exit of this routine ecx store the correct number.

Ok...I don't explain how to inject new code in a section...I assume that who reads this tut knows how to make this...hum...Micro\$oft coders can click on start button on the taskbar and then click on 'help' ☺

The code above is executed everytime we create a new nodes...now we need to see where the program read/write pointers when it deletes a node. Always with the bpm set above we delete a node... Since breaks first in the same routine already seen above...and after it breaks a second time here:

edx = position of the pointer to delete (see above)

:004543F1	<pre> lea eax, [eax+00] push ebx mov ebx, eax xor eax, eax mov [edx*4+ebx+04], eax mov eax, [ebx+20] mov ecx, eax dec ecx cmp edx, ecx jge 0045441C sub eax, edx dec eax mov ecx, eax shl ecx, 02 lea eax, [edx*4+ebx+8] lea edx, [edx*4+ebx+4] call 00402784 dec dword ptr [ebx+20] pop ebx ret </pre>	<pre> eax = 0 replace the pointer in the array with 0 eax = number of pointers ecx = eax number of pointers - 1 </pre>
-----------	---	--

❶

This routine deletes a pointers in an array for the selected node...but not only...to explain better what happen here I've broken this piece of code in four part (colours)...

- First the routine uses the position associated at the pointers (node) to replace it in the array with 0000 (deleted).
- Here the routine compares the position of the deleted pointer (edx) with the position of the last pointer in the array (ecx). If edx is greater (?) or equal at ecx means that the pointer deleted was the last in the array, and the code jump to ❶, where decrement the number of the pointer in this array.

If the compare instruction above says that the position of the deleted pointer is lower than the position of the last pointer in the array, this means that the deleted pointer is not the last in the array.

If we check in the array there's a 'hole' of 0000 where the pointer is been deleted. What the code here does is calculates how pointers there are from this 'hole' at the last pointer included...why?? Simple...to fill this 'hole' moving all these pointers one position (dword) back...

This job is done by the call 0042784...when the code calls this routine...ecx = number of byte to move back of 1 dword in the array, edx = offset in array where to move back the pointers, eax = offset in the array of the first pointer to move back. When the job is done the code jump to ❶

Dec the number of the pointers stored in the array.

Before to show how we can patch this routine...is important to remember that...the position of a pointer is the number of that pointer -1, the position is zero base...for ex. If the code refers to the pointer with position 3, this pointer in the array is the 4th ...and so on... the routine above uses the position of a pointer to delete it!!

We need to remember too, that in our new arrays, the first 7 pointers are stored in the correct original position, but the pointers from 8th to up are stored 5 dwords after the 7th pointer (see 3 pages above). Ok...with this in mind...

:xxxxxxx	lea eax, [eax+00] push ebx mov ebx, eax xor eax, eax push edx cmp edx, 6 jle xxxxxxx1 add edx, 5	eax = 0 save original position for the pointer to delete
:xxxxxxx1	mov [edx*4+ebx+04], eax pop edx mov eax, [ebx+20] mov ecx, eax dec ecx cmp edx, ecx jge xxxxxxx2 sub eax, edx dec eax mov ecx, eax shl ecx, 02 cmp edx, 6 jle xxxxxxx3 add edx, 5	fix the position if the pointer to delete is over 7 replace the pointer in the array with 0 original position for the pointer to delete eax = number of pointers ecx = eax number of pointers -1
:xxxxxxx2	lea eax, [edx*4+ebx+8] lea edx, [edx*4+ebx+4] call 00402784 dec dword ptr [ebx+20] pop ebx ret	Jmptonew routine if deleted pointer is 'le' 6 fix the position if the pointer to delete is over 7
:xxxxxxx3	lea eax, [edx*4+ebx+8] mov ecx, 6 sub ecx, edx lea edx, [edx*4+ebx+4] mov esi, eax mov edi, edx	

<pre> rep movsd inc ecx lea esi, [ebx+34] rep movsd mov ecx, [ebx+20] dec ecx sub ecx, 7 add edi, 14 rep movsd jmp xxxxxxxx2 </pre>	<p>move the pointers below the 8th, 1 dword back</p> <p>move the pointers above the 8th, below the 5 dwords (pointers at others info). The num of pointers to move below is (6-position of the deleted pointer)</p> <p>move the pointers above the 8th, 1 dword back</p>
--	---

...I know...I've lost someone here...☺...but the new code (blue) is not difficult to understand...I don't want waste your time...but it's important that we understand how it works ...thus better if you examine every lines of code.

Great!!...now we have to see only what happen when the proggy save our marvellous tree ...
ok...like above with our bpm on the 7th pointer in the array...we save all...and Sice breaks here:

```
:00454444      mov eax, [edx*4+eax+4]
```

Few things to say here, after this line of code eax contains the pointer at the position stored in edx...
We need again to insert a call at our new code...

```

:xxxxxxx      cmp edx, 6
               jle xxxxxxxx1
               add edx, 5
:xxxxxxx1     mov eax, [edx*4+eax+4]
               ret

```

...that's all...we have changed all critical pieces of code...and the proggy is happy...and a little fatter than before ☺....

Well...we are at the end...I know someone has hoped for this moment from the second page ...but who is arrived until here...now he's happy...I'VE SAID HAPPY!!!! ...MUST BE HAPPY...because I've lost 3 nights of wild sex with my girlfriend to write this!!!! ☺☺☺
Jokes a part I hope that someone have learnt something in more about this fantastic reversing world...

Thanks at all ...see ya in next tut ...

CIAO

MaV3RiCk

maverickluke@hotmail.com

- Please...don't send me emails...
- ...with cracking requests...
- ...with a request for this crack...all info here are enough...but I think that 29\$ is a good price for this little program...the coder that sell it doesn't steal your money if you buy it!!!