When memory management goes bad...

Author: ReWolf
Contact: rewolf [a t] rewolf.pl
http://rewolf.pl

1. Introduction

Let's imagine some simple garbage collection (GC) mechanism based on a singly-linked list. It will be as simple as possible, so it will remember head of the list and a number of stored elements. Each element will contain size of the allocated memory, allocated buffer and of course address of the next element. Actually it will be pseudo-GC, because there won't be any automatic memory freeing, so garbage collection will be triggered manually under some circumstances. GC interface will provide three main functions for memory operations:

- *gc_alloc()*
- *gc_realloc()*
- *gc_free()*

and two functions for GC maintaining:

- *gc_getNumberOfElements()*
- *gc_freeGCMemory()*

Usage of such GC is very intuitive, at the beginning a developer will call *gc_getNumberOfElements()* and remember returned value (I'll call it gc_start), then he can allocate (reallocate) memory with *gc_alloc()* (*gc_realloc()*) as many times as he want and even if sometimes he will not use *gc_free()* (sic!), nothing very bad will happen (but it could ;)). After finishing particular part of code *gc_freeGCMemory()* should be called to free remained memory. I'll show it on a simple pseudo-code:

```
...
int gc_start = gc_getNumberOfElements();
...
//processing some complicated commands, with many
//memory allocations
...
gc_freeGCMemory(gc_start);
...
```

Now you can imagine what can happen if someone will not predict how many memory allocations a specific part of code will use, and he will only rely on the *gc_freeGCMemory()*. I'm writing about this, because I found...

2. Real life example...

Yes, there is at least one example of such mechanism and it is implemented in application that is rather commonly used by windows users: cmd.exe.

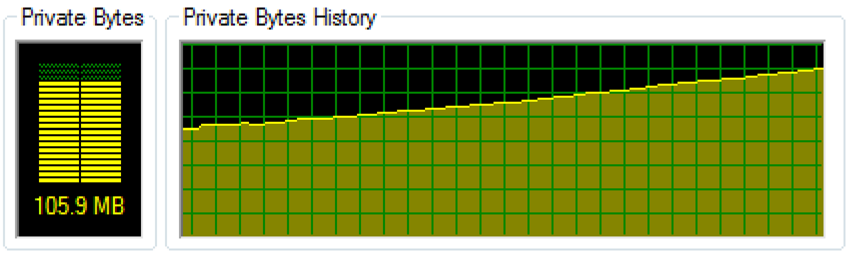
Few months ago I've encountered a strange behavior when I wanted to list some big collection of files and execute some command on every single file. I used for this task 'for' command similar to this:

```
for /R c:\ %c in (*.*) do echo %c >> cmd_ptc.log
```

After few hours of processing I get a very disturbing message:

```
“Not enough storage is available to process this command.”
```

Actually it is system error code 0x8 (*ERROR_NOT_ENOUGH_MEMORY*) translated by *FormatMessage* API. I've done some more tests looking at an application memory usage in Process Explorer:



It can eat all memory available for the process (and it is not memory allocated for a screen buffer as some of you may think). I decided to check why cmd.exe uses such amount of memory, and if it will be possible fix it, because I'm used to automate some tasks with it and I'm still not convinced to PowerShell. Question *'why it uses such amount of memory?'* was already answered (at least partially, because besides this carelessly used pseudo-GC, I found also two things that looks like obvious bugs), now I can start fixing.

3. Cmd.exe internals

Cmd.exe uses a pseudo-GC mechanism described in the first section, probably to avoid some accidental memory leaks during processing of complicated commands. I didn't reviewed all the code, but only fragments related to processing of 'for' command. From what I saw, I can say that most memory allocations are freed after finishing each command, for example:

```
c:\>echo “Hello World”
* enable GC
* do operations required by requested command
* print result
“Hello World”
* free all GC memory allocated since 'enable GC'
c:\>
```

All my research were done on the cmd.exe taken from Windows Vista Home Premium x86 SP2 Eng:

```
md5         : 74f26fc01b180d4a99a168ed69c30a53
file size   : 318976 bytes
file version: 6.0.6001.18000
file date   : 2008-01-21 04:23
```

In further paragraphs I'll refer to offsets (raw, rva) in this specific cmd.exe version, but all changes can be probably easily applied to other versions, because I'll be also referring to names of the functions taken from PDB symbols.

GC related functions from cmd.exe:

- mkstr() ↔ gc_alloc():

```
      wchar_t *__stdcall mkstr(unsigned int a1);
      @ .text:4AD01E3D

      a1    : the number of bytes to be allocated
      return: address of allocated memory buffer
```

- resize() ↔ gc_realloc():

```
wchar_t *__stdcall resize(wchar_t *a1, unsigned int a2);
@ .text:4AD020F1

a1    : address of memory buffer that will be resized
a2    : new buffer size
return: address of new memory buffer
```

- FreeStr() ↔ gc_free():

```
void __stdcall FreeStr(wchar_t *a1);
@ .text:4AD01773

a1    : address of memory buffer that will be freed
```

- DCount ↔ gc_getNumberOfElements():

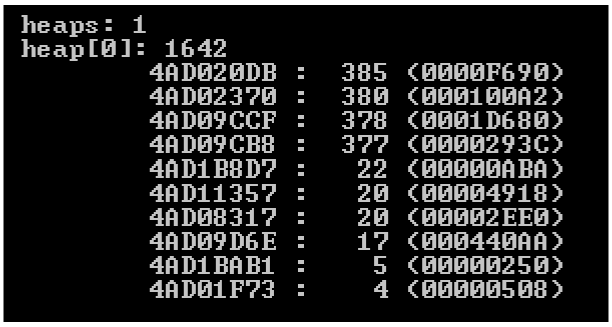
```
DWORD DCount;
@ .data:4AD240BC
```

- FreeStack() ↔ gc_freeGCMemory():

```
void __stdcall FreeStack(unsigned int a1);
@ .text:4AD03031

a1    : (gc_getNumberOfElements() - a1) number of elements will be
        removed (freed) from the list
```

To determine possible leaks I've wrote a simple tool that monitors all heap allocations, and prints all active memory regions (not freed) sorted by: number of allocations, return address and overall size of allocated memory for each return address. There were of course some 'hacks' to show return addresses of *mkstr()* and *resize()* functions instead of *HeapAlloc()* and *HeapReAlloc()*. A sample output produced by this tool looks like this:



```
heaps: 1
heap[0]: 1642
        4AD020DB :  385 (0000F690)
        4AD02370 :  380 (000100A2)
        4AD09CCF :  378 (0001D680)
        4AD09CB8 :  377 (0000293C)
        4AD1B8D7 :   22 (00000ABA)
        4AD11357 :   20 (00004918)
        4AD08317 :   20 (00002EE0)
        4AD09D6E :   17 (000440AA)
        4AD1BAB1 :    5 (00000250)
        4AD01F73 :    4 (00000508)
```

An interpretation of above is rather simple, first column represent return address, second is number of separate allocations, and third is size of allocated memory. With this tool I was able to determine few places where I can start my research. Further analysis under OllyDbg and IDA revealed me exact names of functions with problematic allocations:

– _FindFixAndRun @ 0x4AD021F7
– _ECWork @ 0x4AD04292
– _FLoopWork @ 0x4AD10C04

– _FRecurseWork @ 0x4AD1B855

In the next paragraphs I'll describe all modifications that should be done in each of mentioned functions. Most patches will be illustrated with pseudo-C listings, with some references to an assembly to show what exactly was patched. Showing all modification only on an assembly listing would probably mess all ideas and lecture wouldn't be as straightforward as it is now.

4. FindFixAndRun

This function is responsible for parsing and verifying command and calling specific function for each batch command. Offsets to functions are obtained through *FindCmd()* (@ 0x4AD023B1) and *GetFuncPtr()* (@ 0x4AD03271). *FindFixAndRun()* can cause two leaks, first is easy to fix, second is a bit more complicated. The easiest one is caused by call to *GetTitle()* (@ 0x4AD02329) function:

```
v1 = GetTitle(a1);                      //push    ebx
                                        //call    _GetTitle@4
if ( v1 )                               //test    eax, eax
                                        //jz      short loc_4AD019C0
      SetConTitle(v1);                  //push    eax
                                        //call    _SetConTitle@4
```

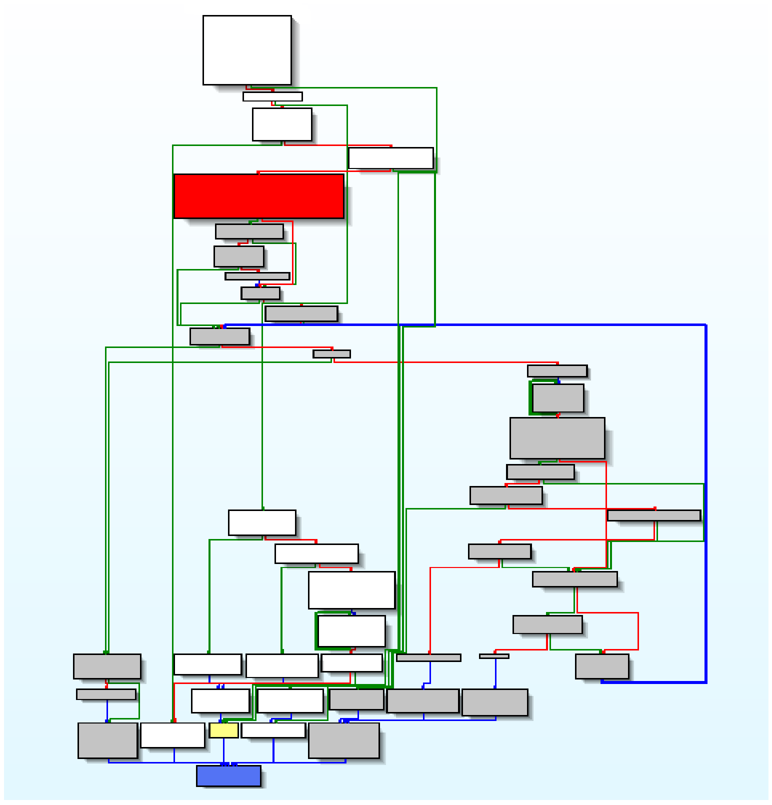
GetTitle() allocates memory with mkstr() function and it should be freed if we don't need it anymore. The variable v1 is used only in this fragment of function, so it can be freed after call to *SetConTitle()*:

```
v1 = GetTitle(a1);                      //push    ebx
                                        //call    _GetTitle@4
if ( v1 )                               //test    eax, eax
{                                       //jz      short loc_4AD019C0
      v2 = v1;                          //push    eax
      SetConTitle(v1);                  //push    eax
                                        //call    _SetConTitle@4
      FreeStr(v2);                      //call    _FreeStr@4
}
```

As you can see, I'm putting eax on the stack, before call to *SetConTitle()* and after call, I'm just calling *FreeStr()*, which takes this eax from the stack.

`Note: Of course in the executable, I cannot just insert instruction like this, so I've to put an extra jump after first push and put rest of the code in some free space at the end of code section, but it is rather obvious and I'll silently skip such details. All modifications will be listed in details at the end of this article, there will be also available a patched binary for download.`

Second leak is caused by call to *TokStr()* (@ 0x4AD01F27), which calls *gresize()* or *gmkstr()* functions. Those functions are similar to *resize()* and *mkstr()*, but in case of memory allocation failure they are calling *Abort()* procedure. To remove this leak I need to remember a value returned by *TokStr()*, and call *FreeStr()* (with this remembered value) at the end of the function, but only for the paths of execution that contains call to *TokStr()*. Sounds complex, but it is doable. There is also an easier solution, I can add variable to the stack frame and initialize it with zero, then I can assign to this variable value returned by *TokStr()* and call *FreeStr()* just before function end. *FreeStr()* handles zero as an argument, and just returns without any actions. The easiest way came to my mind after I've done this patch, but I'm utilizing similar method in *FLoopWork()* fix. To design this patch I've used 'graph view' in IDA. At first I've set different color for nodes that belongs to chosen execution path, then I've started grouping some nodes to simplify function graph.



- red node contains call to *TokStr()*
- gray nodes belongs to execution path of my interest
- white nodes (those at the bottom) can be grouped to simplify view
- blue node is the function return
- yellow node can be reached from white and gray nodes and it needs special handling

I'll remember result from the *TokStr()* on the stack with just simple push:

Original code	Patched code

`push 2`
`lea eax, [ebp+var_224]`
`push eax`
`push dword ptr [ebx+3Ch]`
`call _TokStr@12`
`mov esi, eax`
 	
`cmp edi, 0Ah`
	`push 2`
	`lea eax, [ebp+var_224]`
	`push eax`
	`push dword ptr [ebx+3Ch]`
	`call _TokStr@12`
	`mov esi, eax`
	`push eax`
	`cmp edi, 0Ah`

Now I need to add call to *FreeStr()* at the places where gray nodes connects with blue node and where gray node connects yellow node. There are two such places for gray-blue and one gray-yellow:

Original code	Patched code
`0x4AD019D1 (gray-blue)`	

`mov dword ptr [ebp-4], -2`
`call sub_4AD032D9`
	`mov dword ptr [ebp-4], -2`
	`call sub_4AD032D9`

`mov     eax, [ebp-440h]`
`call    __SEH_epilog4_GS`
`retn    4`
	`call    _FreeStr@4`
	`mov     eax, [ebp-440h]`
	`call    __SEH_epilog4_GS`
	`retn    4`

`0x4AD16B6E (gray-blue)`	

`call    _PutStdErr`
`pop     ecx`
`pop     ecx`
`xor     eax, eax`
`inc     eax`
`mov     _LastRetCode, eax`
 	
`call    __SEH_epilog4_GS`
`retn    4`
	`call    _PutStdErr`
	`pop     ecx`
	`pop     ecx`
	`xor     eax, eax`
	`inc     eax`
	`mov     _LastRetCode, eax`
	`xchg    eax, [esp]`
	`push    eax`
	`call    _FreeStr@4`
	`pop     eax`
	`call    __SEH_epilog4_GS`
	`retn    4`

`0x4AD0229A (gray-yellow)`	

`push    esi`
`push    edi`
`call    _CheckHelpSwitch@8`
`test    al, al`
`jnz     loc_4AD06E11`
`...`
`loc_4AD06E11:`
 	
`xor     eax, eax`
`inc     eax`
	`push    esi`
	`push    edi`
	`call    _CheckHelpSwitch@8`
	`test    al, al`
	`jnz     loc_4AD233B8`
	`...`
	`loc_4AD233B8:`
	`call    _FreeStr@4`
	`xor     eax, eax`
	`inc     eax`

After those changes function shouldn't leak memory anymore.

5. FLoopWork

Changes in this function will be slightly different than in *FindFixAndRun()* because they will fix one obvious bug and one not obvious bug which I'm not quite sure if it is really bug or just some weird design (but this patch drastically decreases memory usage). I've also removed two inlined *wcslen()* to get some more free space in the code section for a future use (finally I didn't used this free space, but I left it in the executable).

Let's start from the obvious one. Below pseudo-code shows some very basic way to reduce a number of the memory allocations in some loop:

```
baseStr = "base";
baseSize = strlen(baseStr) + 1;
curSize = baseSize;
buffer = alloc(curSize);
strcpy(buffer, baseStr);
do
{
    addStr = getNextStr();           //returns next string
    addSize = strlen(addStr);
    if (curSize < baseSize + addSize)
    {
        curSize = baseSize + addSize;
        buffer = realloc(buffer, curSize);
    }
    strcat(buffer, addStr);
    printf(buffer);
    buffer[baseSize - 1] = 0;
}
while (some_condition);
```

The idea is very simple, memory is reallocated only if a bigger buffer is needed, else program will use already allocated buffer. Almost identical mechanism is used in *FLoopWork()*, except one small thing that makes all 'optimization' useless. Apparently someone wrote this code in hurry, because in cmd.exe condition that checks if there is sufficient memory looks like this:

```
        if (curSize < curSize + addSize)
        {
            curSize = curSize + addSize;
            buffer = realloc(buffer, curSize);
        }
```

As you can see, this condition is always true, and memory is reallocated on every loop pass. Beside this, curSize is increased every-time with addSize value, which means that at the end of the loop buffer will be large enough to hold not only baseStr+addStr, but baseStr and all addStr generated during execution of this loop. In below table you can find patches that I've done to remove this bug:

Original code	Patched code
0x4AD10C06: - added one local variable to the stack frame	

`push ebp`
`mov ebp, esp`
`sub esp, 280h`
	`push ebp`
	`mov ebp, esp`
	`sub esp, 284h`

0x4AD11427: - removed one inlined call to wcslen() - assigning length of given string to the added variable [ebp-284h]	

`loc_4AD11427:          ;\`
`  mov   dx, [ecx]      ; \`
`  inc   ecx            ; |`
`  inc   ecx            ; | inlined`
`  test  dx, dx         ; | wcslen()`
`  jnz   loc_4AD11427   ; |`
`  sub   ecx, esi       ; /`
`  sar   ecx, 1         ;/`
`  jz    loc_4AD106E8   ; > Abort()`
`  lea   edx, [eax+2]`
`loc_4AD1143E:          ;\`
`  mov   cx, [eax]      ; \`
`  inc   eax            ; | second`
`  inc   eax            ; | inlined`
`  test  cx, cx         ; | wcslen()`
`  jnz   loc_4AD1143E   ; |`
`  sub   eax, edx       ; /`
`  sar   eax, 1         ;/`
`  lea   ebx, [eax+1]`
`  lea   eax, [ebx+ebx]`
`  push  eax`
`  call  _mkstr@4`
	`loc_4AD11427:          ;\`
	`  mov   dx, [ecx]      ; \`
	`  inc   ecx            ; |`
	`  inc   ecx            ; | inlined`
	`  test  dx, dx         ; | wcslen()`
	`  jnz   loc_4AD11427   ; |`
	`  sub   ecx, esi       ; /`
	`  sar   ecx, 1         ;/`
	`  jz    loc_4AD106E8   ; > Abort()`
	`  mov   eax, ecx`
	`  mov   [ebp-284h], ecx`
	`  lea   ebx, [eax+1]`
	`  lea   eax, [ebx+ebx]`
	`  push  eax`
	`  call  _mkstr@4`

0x4AD11318: - removed one inlined call to wcslen() - using remembered length of string from [ebp-284h]	

```
loc_4AD11318:          ;\                loc_4AD11318:          ;\
  mov     cx, [eax]    ; \                 mov     cx, [eax]    ; \
  inc     eax          ; |                 inc     eax          ; |
  inc     eax          ; | inlined         inc     eax          ; | inlined
  test    cx, cx       ; | wcslen()        test    cx, cx       ; | wcslen()
  jnz     loc_4AD11318 ; |                 jnz     loc_4AD11318 ; |
  sub     eax, edx     ; /                 sub     eax, edx     ; /
  sar     eax, 1       ;/                  sar     eax, 1       ;/
  add     eax, ebx                         mov     ecx, eax
                                           add     eax, [ebp-284h]
                                           inc     eax
  cmp     ebx, eax                         cmp     ebx, eax
  jnb     loc_4AD11361                     jnb     loc_4AD11361
  lea     eax, [ebp-228h]
  lea     edx, [eax+2]
loc_4AD11335:          ;\
  mov     cx, [eax]    ; \
  inc     eax          ; | second
  inc     eax          ; | inlined
  test    cx, cx       ; | wcslen()
  jnz     loc_4AD11335 ; |
  sub     eax, edx     ; /
  sar     eax, 1       ;/
  add     ebx, eax                         add     ebx, ecx
  test    edi, edi                         test    edi, edi
  lea     eax, [ebx+ebx]                   lea     eax, [ebx+ebx]
  push    eax                              push    eax
  jz      loc_4AD143ED                     jz      loc_4AD143ED
  push    edi                              push    edi
  call    _resize@8                        call    _resize@8
```

Now it is time for the second problem in *FLoopWork()*. For better understanding I'll first describe *ForFree()* (@ 0x4AD09C5A) function:

```
int ForFree(int a1)
{
    if ( a1 )
        FreeStack(a1);
    else
        a1 = DCount;
    return a1;
}
```

Basically it is just a helper for the *FreeStack()*, the only difference is that it will return the *DCount* (a number of allocated memory regions) value if passed argument is equal to zero. It is very convenient to use it in any kind of loops, for example:

```
int baseMR = ForFree(0);
do
{
    complicatedFunction01(x, y, z);
    complicatedFunction02(x, y, z);
    baseMR = ForFree(baseMR);
}
while (some_condition)
```

Such construction guarantee that all memory allocated during each iteration will be freed. Of course it could be done without defining *ForFree()*, but as I'm not clairvoyant, I'll not try to explain why it is done in the separate function. Let's back to the merits of the case, the main reason why I'm talking about it is the way how *ForFree()* is used in cmd.exe:

```
int baseMR = 0;
do
{
    complicatedFunction01(x, y, z);
    complicatedFunction02(x, y, z);
    baseMR = ForFree(baseMR);
}
while (some_condition)
```

This relatively small change causes that memory allocated on the first iteration will not be freed. *ForFree()* is used four times in cmd.exe (once in *FParseWork()*, once in *eFor()* and twice in *FLoopWork()*) , and in all four occurrences it is used in this 'buggy' way. I've patched only one occurrence in *FLoopWork()* and it was sufficient to reduce memory usage to the acceptable level. Proposed patch adds *ForFree(0)* at the beginning of every iteration, it could be done better, but it was done as a proof of concept rather than real solution for mass usage.

Original code	Patched code
`0x4AD10D79`	
`call     _ffirst@16`	`call     _ffirst@16`
`test     al, al`	`test     al, al`
`jz       loc_4AD10D29`	`jz       loc_4AD10D29`
	`push     0`
	`call     _ForFree@4`
	`mov      [ebp+var_25C], eax`
`mov      esi, [ebp+var_270]`	`mov      esi, [ebp+var_270]`
`mov      eax, [ebp+lpFileName]`	`mov      eax, [ebp+lpFileName]`

6. ECWork and FRecurseWork

Patches in those functions are very similar to modifications in *FindFixAndRun()*, there are some memory buffers allocated through *GetTitile()*, *mkstr()* or *resize()* and I'm just freeing those buffers, so I'll only put all patches together in the table, just for review.

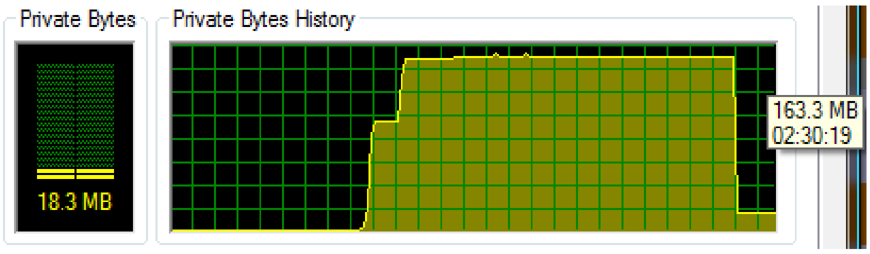
Original code	Patched code
`0x4AD04322 (ECWork)`	
`call     sub_4AD046B8`	`call     sub_4AD046B8`
	`push     ebx`
	`call     _FreeStr@4`
`mov      eax, [ebp+var_230]`	`mov      eax, [ebp+var_230]`
`call     __SEH_epilog4_GS`	`call     __SEH_epilog4_GS`
`retn     0Ch`	`retn     0Ch`
`0x4AD07ECD (ECWork)`	
`call     sub_4AD07F13`	`call     sub_4AD07F13`
	`push     ebx`
	`call     _FreeStr@4`
`mov      eax, [ebp+var_230]`	`mov      eax, [ebp+var_230]`
`call     __SEH_epilog4_GS`	`call     __SEH_epilog4_GS`

`retn    0Ch`	`retn    0Ch`
0x4AD10659 (ECWork)	
`push    eax`	`push    eax`
`call    _PutStdErr`	`call    _PutStdErr`
	`push    ebx`
	`call    _FreeStr@4`
`add     esp, 0Ch`	`add     esp, 0Ch`
`xor     eax, eax`	`xor     eax, eax`
`inc     eax`	`inc     eax`
`call    __SEH_epilog4_GS`	`call    __SEH_epilog4_GS`
`retn    0Ch`	`retn    0Ch`
0x4AD153A5 (ECWork)	
`pop     ecx`	`pop     ecx`
`pop     ecx`	`pop     ecx`
	`push    ebx`
	`call    _FreeStr@4`
`xor     eax, eax`	`xor     eax, eax`
`inc     eax`	`inc     eax`
`call    __SEH_epilog4_GS`	`call    __SEH_epilog4_GS`
`retn    0Ch`	`retn    0Ch`
0x4AD1063F (ECWork)	
`cmp     eax, 3`	`cmp     eax, 3`
`jz      loc_4AD15396`	`jz      loc_4AD23367`
`...`	`...`
`loc_4AD15396:`	`loc_4AD23367:`
	`push    ebx`
	`call    _FreeStr@4`
`mov     eax, edx          ; edx = 1`	`xor     eax, eax`
	`inc     eax`
`call    __SEH_epilog4_GS`	`call    __SEH_epilog4_GS`
`retn    0Ch`	`retn    0Ch`
0x4AD1538C (ECWork)	
`call    _ChangeDir2@8`	`call    _ChangeDir2@8`
	`push    eax`
	`push    ebx`
	`call    _FreeStr@4`
	`pop     eax`
`call    __SEH_epilog4_GS`	`call    __SEH_epilog4_GS`
`retn    0Ch`	`retn    0Ch`
0x4AD1B959 (FRecurseWork)	
`call    _FLoopWork@20`	`call    _FLoopWork@20`
`mov     edi, eax`	`mov     edi, eax`
	`pusha`
	`push    dword ptr [ebp-25Ch]`
	`call    _FreeStr@4`
	`popa`
`mov     eax, [ebp-260h]`	`mov     eax, [ebp-260h]`
`lea     ecx, [eax+2]`	`lea     ecx, [eax+2]`

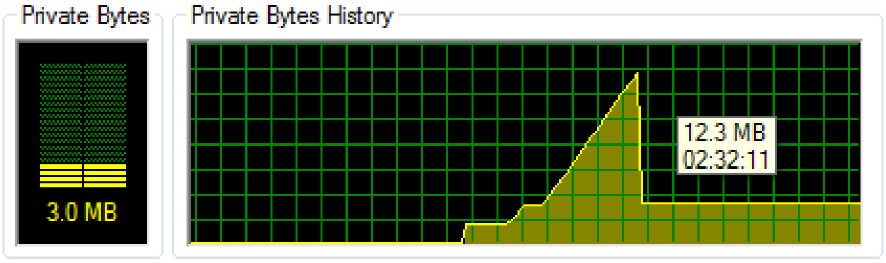
7. Conclusion

There are still many functions that can be 'fixed' in the same manner, but it was not the point of this research. I've achieved my goals - reducing memory usage during execution of the 'for' command. I didn't reported those 'leaks' to Microsoft, because I do not consider those bugs as serious and to be honest I don't believe that someone will care about it. From the quick overview I can confirm that described behavior occurs in probably all x86 versions of cmd.exe (I've looked at: vista sp2, xp sp2, xp sp3 and win7). Below you can compare two memory usage graphs generated during execution of this command:

```
for /R c:\windows\winsxs %c in (*.*) do echo %c
```



Original cmd.exe (maximal memory usage: 163,3 MB)



Patched cmd.exe (maximal memory usage: 12,3 MB)

8. Timeline

```
–  Feb 2009 – discovered problem
–  Jul 2009 – researched and patched cmd.exe
–  Mar 2010 – finished this paper
```