WinDbg: trivial examining of the stack

revision: 1
author: Alexandre Borges

Hello readers, how are you? It has passed a long time since my last post. As I've explained previously, my time for writing is almost zero and it is very hard to write something. Anyway, this is a quick post about WinDbg and stack's dynamic. During my lectures about malware analysis, I am not usually concerned with my audience because most professionals are from security area and malware analysts, so technical explanations should be piece of cake.

Nevertheless, two days ago, I was surprised during a presentation about a malware where I was using WinDbg and showing stack details. Everything was going well until I have made a simple question about arguments of a function and for my surprise I realized that nobody was able to manage the interpretation of the WinDbg's output.

To help everyone, I have decided to write this simple post. Probably, you already know about it, but just in case you also have minor questions, so eventually the subject worth to be revised.

Please, pay attention: colors are important throughout this article!

Trying to be straight, I have written a simple (and not optimized) C program (named debug3.c) without any frills and compiled it using **lcc compiler** (https://www.cs.virginia.edu/~lcc-win32/) on a Win x86 environment. I could have used a Win x64 system, but I would have to explain other details, which they would make the explanation a bit more complicated in this time.

It follows a trivial program and, as you will see, it has very strange parts because I tried to make odd decisions preparing the path for next articles. Some variables were declared, but they were not used because this kind of explanation is not our purpose for while:

```
#include
int function2(s,t,u)
{
int e;
char *myarray2[50]="\n\nThis is the function2.";
printf(*myarray2);
printf("\n\nValues of arguments are: %d %d %d", s, t, u);
printf("\n\nEnter a letter to return to function 1: ");
```

```
scanf("%e", &e);
return 0;
}

int function1(a,b,c)
{
int f;
int m=4;
int n=5;
int o=6;
char *myarray1[50]="\n\nThis is the function1.";
printf(*myarray1);
printf("\n\nValues of arguments are: %d %d %d", a, b, c);
function2(m,n,o);
fflush(stdin);
printf("\n\nEnter a letter to return to function main: ");
scanf("%f", &f);
return 0;
}

int main()
{
int d;
int x=1;
int y=2;
int z=3;
char *myarray0[50]="\n\nThis is the main function.";
printf(*myarray0);
function1(x, y, z);
fflush(stdin);
printf("\n\nWe returned to function main. Enter a letter to return to finish: ");
scanf("%d", &d);
return 0;
}
```

Compiling this program by using **lcc** (believe me, it is easy) results in a program named debug3.exe.

As you should know, when we list a function in assembly language, it normally has a prologue (unless it is a naked function) as shown below:

```
004012d4 55              push   ebp
004012d5 89e5            mov    ebp,esp
004012d7 81eccc000000    sub    esp,0CCh  // saving space for local variables.
```

Based on previous concepts, we know few facts about these three instructions:

a. At first instruction, it is pushing the EBP (saved frame buffer) for using later after returning from this current function.
b. At second instruction, it is making the current ESP as the new EBP.
c. The content of the EBP takes us to the previous EBP. Thus, we could follow backward until the start of the debugger's calling.
d. The return pointer (RET) to previous function is equal to EBP + 4. (**RET = EBP + 4)**.
e. The first argument used by the current function starts in EBP + 8 (start of arguments = **EBP + 8**).
f. Function parameters are given by **EBP + value** .
g. Local variables of functions are given by **EBP - value** .
h. Windows usually does not allocate the whole stack at first time, but it does it on demand.
i. The **TEB (Thread Environment Block)** is given by **FS:[0].**

Once WinDbg is configured (I really hope you know how to do it), execute:

```
0:000> .restart
CommandLine: C:\lcc\projects\debug3\lcc\debug3.exe
Symbol search path is: srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 0040d000 c:\lcc\projects\debug3\lcc\debug3.exe
ModLoad: 7c900000 7c9af000 ntdll.dll
ModLoad: 7c800000 7c8f6000 C:\WINDOWS\system32\kernel32.dll
ModLoad: 7e410000 7e4a1000 C:\WINDOWS\system32\USER32.DLL
ModLoad: 77f10000 77f59000 C:\WINDOWS\system32\GDI32.dll
ModLoad: 73d90000 73db7000 C:\WINDOWS\system32\CRTDLL.DLL
(a20.8a4): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffde000 ecx=00000005 edx=00000020 esi=00241f48 edi=00241eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc int 3
```

List the current breakpoints by executing the following command:

```
0:000> bl
```

Of course, we do not have breakpoints yet. So, it would be nice to create few ones right at each call (main, function1 and function2), which they will be kept even after restarting the WinDbg. Thus, instead of using the "bp" command, let's use "bu" command:

0:000> **bu debug3!main**
0:000> **bu debug3!function1**
0:000> **bu debug3!function2**

Listing the breakpoints again, we have:

0:000> **bl**
0 e 004013fe 0001 (0001) 0:**** debug3!main
1 e 0040134f 0001 (0001) 0:**** debug3!function1
2 e 004012d4 0001 (0001) 0:**** debug3!function2

If you make a mistake, so you should remember that **bp *breakpointID*** (for example, **bp 2**) deletes the specified breakpoint.

Run the program until it hit the third breakpoint at function2 call:

0:000> **g**
Breakpoint 0 hit
eax=0040b030 ebx=7ffde000 ecx=00409058 edx=00000000 esi=00bcf754 edi=00bcf6ee
eip=004013fe esp=0012ff74 ebp=0012ffc0 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
debug3!main:
004013fe 55 push ebp

0:000> **g**
Breakpoint 1 hit
eax=0000001c ebx=7ffde000 ecx=7c810ea6 edx=0040b340 esi=0040b2f8 edi=0012ff64
eip=0040134f esp=0012fe80 ebp=0012ff70 iopl=0 nv up ei pl nz ac pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000216
debug3!function1:
0040134f 55 push ebp

0:000> **g**
Breakpoint 2 hit
eax=00000020 ebx=7ffde000 ecx=7c810ea6 edx=0040b3d5 esi=0040b230 edi=0012fe70
eip=004012d4 esp=0012fd8c ebp=0012fe7c iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206

debug3!function2:
004012d4 55 push ebp

As at this point the function2 did not create its respective stack frame (see the last instruction in brown above), so execute more a couple of instructions as follow:

0:000> **p**
eax=00000020 ebx=7ffde000 ecx=7c810ea6 edx=0040b3d5 esi=0040b230 edi=0012fe70
eip=004012d5 esp=0012fd88 ebp=0012fe7c iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
debug3!function2+0x1:
004012d5 89e5 mov ebp,esp

0:000> **p**
eax=00000020 ebx=7ffde000 ecx=7c810ea6 edx=0040b3d5 esi=0040b230 edi=0012fe70
eip=004012d7 esp=0012fd88 **ebp=0012fd88** iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
debug3!function2+0x3:
004012d7 81eccc000000 sub esp,0CCh

Now, check the stack by executing the following command:

0:000> **kb**
ChildEBP RetAddr Args to Child
0012fd88 004013c3 00000004 00000005 00000006 debug3!function2+0x3
0012fe7c 0040145c 00000001 00000002 00000003 debug3!function1+0x74
0012ff70 004012bc 00000001 00144380 001437e8 debug3!main+0x5e
0012ffc0 7c817067 00bcf6ee 00bcf754 7ffde000 debug3+0x12bc
0012fff0 00000000 00401225 00000000 78746341 kernel32!BaseProcessStart+0x23

Watch the current registers by running:

0:000> **r**
eax=00000020 ebx=7ffde000 ecx=7c810ea6 edx=0040b3d5 esi=0040b230 edi=0012fe70
eip=004012d7 esp=0012fd88 **ebp=0012fd88** iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
debug3!function2+0x3:
004012d7 81eccc000000 sub esp,0CCh

After we have examined the last two outputs, we know that:

1. The **current EBP** is **0012fd88**.

2. **IF** the **function2** was called using three arguments, so they are: **00000004 00000005 00000006.** The first one is at function's EBP + 8.
3. **IF** the **function1** was called using three arguments, so they are: **00000001 00000002 00000003.** The first one is at function's EBP + 8.
4. **IF** the **main** function was called using three arguments, so they are: **00000001 00144380 001437e8.** The first one is at function's EBP + 8.
5. The **RET (return) pointer** (EBP + 4) to function1 is: **004013c3**
6. The **RET (return) pointer** (EBP + 4) to main function is: **0040145c**
7. Following all EBPs backward, we have (* means an address, as pointers):
    - **(*0012fd88) = 0012fe7c**
    - **(*0012fe7c) = 0012ff70**
    - **(*0012ff70) = 0012ffc0**
    - **(*0012ffc0) = 0012fff0**
    - **(*0012fff0) = 00000000**

Thus, the most important questions are:

1. Is it possible to follow EBP pointer backward to the beginning?
2. Do these functions (function1, function2 and main) have three arguments each one?

The first question can be easily answered by using DDS (Dump Dwords with Symbols) command as shown below, where L5 requires that five DWORDS are listed. Thus, for the first EBP, we have:

```
0:000> dds 0012fd88 L5   --> we could have used dds @ebp L5 in this case
0012fd88 0012fe7c       --> function1's EBP
0012fd8c 004013c3 debug3!function1+0x74
0012fd90 00000004       --> possible first argument of the function2  (EBP+ 8)
0012fd94 00000005       --> possible second argument of the function2
0012fd98 00000006       --> possible third argument of the function2
```

Using the function1's EBP above (in blue), we have:

```
0:000> dds 0012fe7c L5
0012fe7c 0012ff70       --> main's EBP
0012fe80 0040145c debug3!main+0x5e
0012fe84 00000001       --> possible first argument of the function1  (EBP+ 8)
0012fe88 00000002       --> possible second argument of the function1
0012fe8c 00000003       --> possible third argument of the function1
```

Using the main's EBP (and at same way for other calls), it is possible to reach the first debugger's call:

0:000> **dds 0012ff70 L5**
0012ff70 0012ffc0 --> debug3's EBP
0012ff74 004012bc debug3+0x12bc
0012ff78 00000001 **--> possible first argument of the main function (EBP+ 8)**
0012ff7c 00144380 **--> possible second argument of the main function**
0012ff80 001437e8 **--> possible third argument of the main function**

0:000> **dds 0012ffc0 L5**
0012ffc0 0012fff0
0012ffc4 7c817067 kernel32!BaseProcessStart+0x23
0012ffc8 **00bcf6ee --> possible first argument of the debug3 program (EBP+ 8)**
0012ffcc **00bcf754 --> possible second argument of the debug3 program (EBP+ 8)**
0012ffd0 **7ffde000 --> possible third argument of the debug3 program (EBP+ 8)**

0:000> **dds 0012fff0 L5**
0012fff0 00000000 --> Start of calling (there is not a previous call)
0012fff4 00000000
0012fff8 00401225 debug3+0x1225
0012fffc 00000000
00130000 78746341

Wow! We have shown that is possible to follow the EBP until the beginning (the debugger's call). Once more, let us to remember the stack and registers:

0:000> **kb**
ChildEBP RetAddr Args to Child
0012fd88 004013c3 00000004 00000005 00000006 debug3!function2+0x3
0012fe7c 0040145c 00000001 00000002 00000003 debug3!function1+0x74
0012ff70 004012bc 00000001 00144380 001437e8 debug3!main+0x5e
0012ffc0 7c817067 00bcf6ee 00bcf754 7ffde000 debug3+0x12bc
0012fff0 00000000 00401225 00000000 78746341 kernel32!BaseProcessStart+0x23

0:000> **r**
eax=00000020 ebx=7ffde000 ecx=7c810ea6 edx=0040b3d5 esi=0040b230 edi=0012fe70
eip=004012d7 esp=0012fd88 **ebp=0012fd88** iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
debug3!function2+0x3:
004012d7 81eccc000000 sub esp,0CCh

Now it is time to answer the second question: how can we actually to verify how many arguments each function has?

The answer is at RET pointer! We should highlight that the return address takes the execution back to the middle of the caller function. Nevertheless, if the stack is good (more about it later), so it possible to use this information to list the entire caller function.

For example, according to the stack above, the execution is at byte 3 of the function2 (EIP + 3). Therefore, to list the whole function, we should execute:

```
0:000> uf 004012d7 - 0x3
debug3!function2:
004012d4 55 push ebp
004012d5 89e5 mov ebp,esp
004012d7 81eccc000000 sub esp,0CCh
004012dd b933000000 mov ecx,33h

debug3!function2+0xe:
004012e2 49 dec ecx
004012e3 c7048c5a5afaff mov dword ptr [esp+ecx*4],0FFFA5A5Ah
004012ea 75f6 jne debug3!function2+0xe (004012e2)

debug3!function2+0x18:
004012ec 56 push esi
004012ed 57 push edi
004012ee 8dbd38ffffff lea edi,[ebp-0C8h]
004012f4 8d35a0b04000 lea esi,[debug3!_internalDigitArray+0x2008 (0040b0a0)]
004012fa b9c8000000 mov ecx,0C8h
004012ff f3a4 rep movs byte ptr es:[edi],byte ptr [esi]
00401301 ffb538ffffff push dword ptr [ebp-0C8h]
00401307 e81c780000 call debug3!printf (00408b28)
0040130c 83c404 add esp,4
0040130f ff7510 push dword ptr [ebp+10h]
00401312 ff750c push dword ptr [ebp+0Ch]
00401315 ff7508 push dword ptr [ebp+8]
00401318 68d5b34000 push offset debug3!_internalDigitArray+0x233d (0040b3d5)
0040131d e806780000 call debug3!printf (00408b28)
00401322 83c410 add esp,10h
00401325 68aab34000 push offset debug3!_internalDigitArray+0x2312 (0040b3aa)
0040132a e8f9770000 call debug3!printf (00408b28)
0040132f 83c404 add esp,4
00401332 8dbd34ffffff lea edi,[ebp-0CCh]
00401338 57 push edi
00401339 68a7b34000 push offset debug3!_internalDigitArray+0x230f (0040b3a7)
0040133e e8f7440000 call debug3!scanf (0040583a)
```

```
00401343 83c408 add esp,8
00401346 b800000000 mov eax,0
0040134b 5f pop edi
0040134c 5e pop esi
0040134d c9 leave
0040134e c3 ret
```

It is important to highlight that we have specified the offset 0x3 for listing all instructions since the function2's beginning. If we hadn't done it, our listing would have shown instructions only from the byte 3 onward, as shown below:

```
0:000> uf function2
debug3!function2:
004012d4 55 push ebp
004012d5 89e5 mov ebp,esp
004012d7 81eccc000000 sub esp,0CCh
004012dd b933000000 mov ecx,33h

debug3!function2+0xe:
004012e2 49 dec ecx
004012e3 c7048c5a5afaff mov dword ptr [esp+ecx*4],0FFFA5A5Ah
004012ea 75f6 jne debug3!function2+0xe (004012e2)
....
```

Could we have used the **u** command? This is an essential point. The **uf** command disassembles the whole function since its beginning and it knows where the function ends. In the other hand, the **u** command also disassembles from the indicated address to onward, but it does not know where the function ends, so we have to do an educated guess.

Is the **u** command useless? It is not at all! In this example, we are working on a perfect case where the stack is not corrupted. During crashes (for example, because buffer overflow), the stack is compromised and we do not know whether the provided address comes from a function or not. Additionally, we neither know whether is correct.

See the difficult in using the **u** command below:

```
0:000> u 004012d7 - 3
debug3!function2:
004012d4 55 push ebp
004012d5 89e5 mov ebp,esp
004012d7 81eccc000000 sub esp,0CCh
```

```
004012dd b933000000 mov ecx,33h
004012e2 49 dec ecx
004012e3 c7048c5a5afaff mov dword ptr [esp+ecx*4],0FFFA5A5Ah
004012ea 75f6 jne debug3!function2+0xe (004012e2)
004012ec 56 push esi

0:000> u
debug3!function2+0x19:
004012ed 57 push edi
004012ee 8dbd38ffffff lea edi,[ebp-0C8h]
004012f4 8d35a0b04000 lea esi,[debug3!_internalDigitArray+0x2008 (0040b0a0)]
004012fa b9c8000000 mov ecx,0C8h
004012ff f3a4 rep movs byte ptr es:[edi],byte ptr [esi]
00401301 ffb538ffffff push dword ptr [ebp-0C8h]
00401307 e81c780000 call debug3!printf (00408b28)
0040130c 83c404 add esp,4

0:000> u
debug3!function2+0x3b:
0040130f ff7510 push dword ptr [ebp+10h]
00401312 ff750c push dword ptr [ebp+0Ch]
00401315 ff7508 push dword ptr [ebp+8]
00401318 68d5b34000 push offset debug3!_internalDigitArray+0x233d (0040b3d5)
0040131d e806780000 call debug3!printf (00408b28)
00401322 83c410 add esp,10h
00401325 68aab34000 push offset debug3!_internalDigitArray+0x2312 (0040b3aa)
0040132a e8f9770000 call debug3!printf (00408b28)

0:000> u
debug3!function2+0x5b:
0040132f 83c404 add esp,4
00401332 8dbd34ffffff lea edi,[ebp-0CCh]
00401338 57 push edi
00401339 68a7b34000 push offset debug3!_internalDigitArray+0x230f (0040b3a7)
0040133e e8f7440000 call debug3!scanf (0040583a)
00401343 83c408 add esp,8
00401346 b800000000 mov eax,0
0040134b 5f pop edi

0:000> u
debug3!function2+0x78:
0040134c 5e pop esi
0040134d c9 leave
```

0040134e c3 ret **---> End of function2**
debug3!function1:
0040134f 55 push ebp
00401350 89e5 mov ebp,esp
00401352 81ecd8000000 sub esp,0D8h
00401358 b936000000 mov ecx,36h
0040135d 49 dec ecx

Have you seen what I told you? We have executed the **u** command five times to be able to see the whole function2, but it does not know where the function2 ends. Of course, the **u** command works, but if we can use **uf** command, so we will use it.

Repeating the previous steps above with other functions, we have:

0:000> **kb**
ChildEBP RetAddr Args to Child
0012fd88 004013c3 00000004 00000005 00000006 debug3!function2+0x3
0012fe7c 0040145c 00000001 00000002 00000003 debug3!function1+0x74
0012ff70 004012bc 00000001 00144380 001437e8 debug3!main+0x5e
0012ffc0 7c817067 00bcf6ee 00bcf754 7ffde000 debug3+0x12bc
0012fff0 00000000 00401225 00000000 78746341 kernel32!BaseProcessStart+0x23

0:000> **uf 004013c3 - 0x74**
debug3!function1:
0040134f 55 push ebp
00401350 89e5 mov ebp,esp
00401352 81ecd8000000 sub esp,0D8h
00401358 b936000000 mov ecx,36h

debug3!function1+0xe:
0040135d 49 dec ecx
0040135e c7048c5a5afaff mov dword ptr [esp+ecx*4],0FFFA5A5Ah
00401365 75f6 jne debug3!function1+0xe (0040135d)

debug3!function1+0x18:
00401367 56 push esi
00401368 57 push edi
00401369 c745fc04000000 mov dword ptr [ebp-4],4
00401370 c745f805000000 mov dword ptr [ebp-8],5
00401377 c745f406000000 mov dword ptr [ebp-0Ch],6
0040137e 8dbd2cffffff lea edi,[ebp-0D4h]
00401384 8d3568b14000 lea esi,[debug3!_internalDigitArray+0x20d0 (0040b168)]

```
0040138a b9c8000000 mov ecx,0C8h
0040138f f3a4 rep movs byte ptr es:[edi],byte ptr [esi]
00401391 ffb52cffffff push dword ptr [ebp-0D4h]
00401397 e88c770000 call debug3!printf (00408b28)
0040139c 83c404 add esp,4
0040139f ff7510 push dword ptr [ebp+10h]
004013a2 ff750c push dword ptr [ebp+0Ch]
004013a5 ff7508 push dword ptr [ebp+8]
004013a8 68d5b34000 push offset debug3!_internalDigitArray+0x233d (0040b3d5)
004013ad e876770000 call debug3!printf (00408b28)
004013b2 83c410 add esp,10h
004013b5 ff75f4 push dword ptr [ebp-0Ch]
004013b8 ff75f8 push dword ptr [ebp-8]
004013bb ff75fc push dword ptr [ebp-4]
004013be e811ffffff call debug3!function2 (004012d4)
004013c3 83c40c add esp,0Ch
004013c6 ff3548c14000 push dword ptr [debug3!_translation+0x23f (0040c148)]
004013cc e8b7790000 call debug3!fprintfv+0x19a (00408d88)
004013d1 83c404 add esp,4
004013d4 6860b34000 push offset debug3!_internalDigitArray+0x22c8 (0040b360)
004013d9 e84a770000 call debug3!printf (00408b28)
004013de 83c404 add esp,4
004013e1 8dbd28ffffff lea edi,[ebp-0D8h]
004013e7 57 push edi
004013e8 685db34000 push offset debug3!_internalDigitArray+0x22c5 (0040b35d)
004013ed e848440000 call debug3!scanf (0040583a)
004013f2 83c408 add esp,8
004013f5 b800000000 mov eax,0
004013fa 5f pop edi
004013fb 5e pop esi
004013fc c9 leave
004013fd c3 ret
```

It is nice! We were able to list the whole function1 and we have also confirmed that the function2 was called using three arguments!

By using the same technique for other functions, we have:

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012fd88 004013c3  00000004 00000005 00000006 debug3!function2+0x3
0012fe7c 0040145c  00000001 00000002 00000003 debug3!function1+0x74
0012ff70 004012bc  00000001 00144380 001437e8 debug3!main+0x5e
```

0012ffc0 7c817067 00bcf6ee 00bcf754 7ffde000 debug3+0x12bc
0012fff0 00000000 00401225 00000000 78746341 kernel32!BaseProcessStart+0x23

0:000> **uf 0040145c - 0x5e**
debug3!main:
004013fe 55 push ebp
004013ff 89e5 mov ebp,esp
00401401 81ecd8000000 sub esp,0D8h
00401407 b936000000 mov ecx,36h

debug3!main+0xe:
0040140c 49 dec ecx
0040140d c7048c5a5afaff mov dword ptr [esp+ecx*4],0FFFA5A5Ah
00401414 75f6 jne debug3!main+0xe (0040140c)

debug3!main+0x18:
00401416 56 push esi
00401417 57 push edi
00401418 c745fc01000000 mov dword ptr [ebp-4],1
0040141f c745f802000000 mov dword ptr [ebp-8],2
00401426 c745f403000000 mov dword ptr [ebp-0Ch],3
0040142d 8dbd2cffffff lea edi,[ebp-0D4h]
00401433 8d3530b24000 lea esi,[debug3!_internalDigitArray+0x2198 (0040b230)]
00401439 b9c8000000 mov ecx,0C8h
0040143e f3a4 rep movs byte ptr es:[edi],byte ptr [esi]
00401440 ffb52cffffff push dword ptr [ebp-0D4h]
00401446 e8dd760000 call debug3!printf (00408b28)
0040144b 83c404 add esp,4
0040144e ff75f4 push dword ptr [ebp-0Ch]
00401451 ff75f8 push dword ptr [ebp-8]
00401454 ff75fc push dword ptr [ebp-4]
00401457 e8f3feffff call debug3!function1 (0040134f)
0040145c 83c40c add esp,0Ch
0040145f ff3548c14000 push dword ptr [debug3!_translation+0x23f (0040c148)]
00401465 e81e790000 call debug3!fprintfv+0x19a (00408d88)
0040146a 83c404 add esp,4
0040146d 68fbb24000 push offset debug3!_internalDigitArray+0x2263 (0040b2fb)
00401472 e8b1760000 call debug3!printf (00408b28)
00401477 83c404 add esp,4
0040147a 8dbd28ffffff lea edi,[ebp-0D8h]
00401480 57 push edi
00401481 68f8b24000 push offset debug3!_internalDigitArray+0x2260 (0040b2f8)

00401486 e8af430000 call debug3!scanf (0040583a)
0040148b 83c408 add esp,8
0040148e b800000000 mov eax,0
00401493 5f pop edi
00401494 5e pop esi
00401495 c9 leave
00401496 c3 ret

It is amazing! We have also proved that function1 was called with three arguments at main function.

Again, repeating the technique:

0:000> **kb**
ChildEBP RetAddr Args to Child
0012fd88 004013c3 00000004 00000005 00000006 debug3!function2+0x3
0012fe7c 0040145c 00000001 00000002 00000003 debug3!function1+0x74
0012ff70 004012bc 00000001 00144380 001437e8 debug3!main+0x5e
0012ffc0 7c817067 00bcf6ee 00bcf754 7ffde000 debug3+0x12bc
0012fff0 00000000 00401225 00000000 78746341 kernel32!BaseProcessStart+0x23

0:000> **uf 004012bc - 0x12bc**

Flow analysis was incomplete, some code may be missing
debug3:
00400000 4d dec ebp
00400001 5a pop edx
00400002 90 nop
00400003 0003 add byte ptr [ebx],al
00400005 0000 add byte ptr [eax],al
00400007 000400 add byte ptr [eax+eax],al
0040000a 0000 add byte ptr [eax],al

Surprise! Our technique has failed! However, we can improvise by doing an educated guess (I've chosen 20 bytes, but I would have continued trying until finding the right point):

0:000> **uf 004012bc - 20**
debug3+0x129c:
0040129c 02ff add bh,bh
0040129e d1ff sar edi,1
004012a0 3530b04000 xor eax,offset debug3!_internalDigitArray+0x1f98 (0040b030)

004012a5 ff352cb04000 push dword ptr [debug3!_internalDigitArray+0x1f94 (0040b02c)]
004012ab ff3528b04000 push dword ptr [debug3!_internalDigitArray+0x1f90 (0040b028)]
004012b1 892514b04000 mov dword ptr [debug3!_internalDigitArray+0x1f7c (0040b014)],esp
004012b7 e842010000 call debug3!main (004013fe)
004012bc 83c418 add esp,18h
004012bf 31c9 xor ecx,ecx
004012c1 894dfc mov dword ptr [ebp-4],ecx
004012c4 50 push eax
004012c5 e8a67a0000 call debug3!fprintfv+0x182 (00408d70)
004012ca c9 leave
004012cb c3 ret

Another suprise: the main function does not have any argument, so those arguments that we have seen in the output from **kb** command are fake!

Finally, if the reader wants to examine all calls from any function, he/she can execute the following command (**uf /c *function***):

0:000> **uf /c main**
debug3!main (004013fe)
debug3!main+0x48 (00401446):
call to debug3!printf (00408b28)
debug3!main+0x59 (00401457):
call to debug3!function1 (0040134f)
debug3!main+0x67 (00401465):
call to debug3!fprintfv+0x19a (00408d88)
debug3!main+0x74 (00401472):
call to debug3!printf (00408b28)
debug3!main+0x88 (00401486):
call to debug3!scanf (0040583a)

Once again, this article has explained how to examine the stack on a perfect case where nothing is corrupted. Obviously, the same fundamentals could be used during analysis of a corrupted stack. If I have enough time, I will release a new post about these interesting cases. Stay tuned!

Please, let me know whether you have liked this trivial article. I hope you have a nice day.

Alexandre Borges.

(LinkedIn: http://www.linkedin.com/in/aleborges and Twitter: @ale_sp_brazil).