Digitalk Smalltalk/VS(E) – 32 bit Windows crash analysis

Andreas Rosenberg
APIS Informationstechnologien GmbH
Im Haslet 42, D-93086 Wörth a.d. Donau, Germany

This is an example how to analyse a Smalltalk/V stack dump.

To debug and analyse machine code, you need a debugger. A powerfull and flexibile debugger is WINDBG, which can be obtained from the Microsoft homepage for free.

I will do a small example how to find information about objects and methods.

Start WINDBG and attach it to the VS process (File|Attach Process...) and select VDEVW.EXE.
The process will be stopped after the attach has been performed. You can use this to set breakpoints, but for now, we simply hit “g” to restart the Smalltalk process.

Now let's execute some Smalltalk code that will crash the VM. Type
```
Object new methodDictionaryArray: #( 1 ).
```
and “Show it”.

Your debugger will show something like this:

```
(900.9e0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
1001464c 8b78f8          mov     edi,[eax-0x8]     ds:0023:fffffffb=????????
```

In most cases you need to know which method was active when the crash happened. In our case, we did provoke the exception, but we still can see on the stack, what has happened. This requires some knowledge and experience.

The first step is to find the EBP chain. The EBP chain links the single stack frames. For an example see the stack dump below, where the links are marked. It's rather easy to find the chain, because the values on the stack are in the same range as the address of the stack. Once you find a candidate, you simply look at the address this DWORD points to. If the value at this address also contains an address in the stack range, you found it (most likely).

Dump the stack - it will look like this:

```
0:000> dd esp
0013fd20  000050b0 00000000 100145b0 10072a74
0013fd30  00d687f4 10073738 10070e50 00bb51e7
0013fd40  10070e50 10070e68 00f175f4 100fc1f0
0013fd50  0013fd60 00b71c47 00d62270 10038070
0013fd60  0013fd80 00b71be5 10070e8c 10038040
0013fd70  10038040 10070e68 00dae9cc 10070e50
0013fd80  0013fd9c 00b92acf 10070e74 10070e68
0013fd90  10070e44 00e27780 10051930 0013fdb8

0:000> dd
0013fda0  00bb527f 10070e50 10070e44 10070d34
0013fdb0  00f175f4 100fc1f0 0013fdc8 00bb50cf
0013fdc0  00ee3d38 100fc15c 0013fdd8 00bb504c
0013fdd0  00f2bae0 100fc128 0013fde8 00b8bc45
0013fde0  00d531e8 100fc128 0013fe10 00bb06b2
0013fdf0  100ddc38 00000003 0000000b 100fd730
0013fe00  10070d34 10070cd0 00e71f70 100fc4f4
0013fe10  0013fe28 00b913b0 00000a03 100fc4f4
```

Once you know the frames, you can find out which objects and methods are involved. This requires knowledge about the structure of the stack frames and the Smalltalk objects. Lets examine the first stack frame listed above. The important information is at offset -1 and -2 for each stack frame – there are some exceptions for some special kinds of stack frames. At offset -1 you will find the receiver object, at offset -2 you will find the compiled method.

```
00f175f4  100fc1f0  0013fd60
```

No lets find the information for the values above. Usually you want to know the class of the receiver object. This information is not directly linked, but must be obtained by examining the first element of the method dictionary array (MDA). The pointer to the MDA is at offset -4 (bytewise).

```
0:000> dd 100fc1f0-4
100fc1ec  00e4160c 100578a8 100578b4 10038040
```

Examine the first slot of the MDA:

```
0:000> dd 00e4160c
00e4160c  00e19814 00e49b30 00e4ec4c 00db17cc
```

Now we have a MethodDictionary:

```
0:000> dd 00e19814
00e19814  00000151 00e372e4 00e009ec 213b0607
```

The third slot of a method dictionary contains the class of the object, we want to know.

```
0:000> dd 00e009ec
00e009ec  00e7005c 00e4160c 0000802d 100e17a8
```

To retrieve the name of the class, we must look at the fourth slot:

0:000> db 100e17a8
100e17a8 TextPane.....a..

Finding the method being sent is much easier. The symbol of a compiled method is stored in the fourth slot of the compiled method object.

```
0:000> dd 00f175f4
00f175f4  00ef0848 00bb5141 00e009ec 100ddc38

0:000> db 100ddc38
100ddc38  printIt......a..
```

Finally we now know the class of the receiver and the method being sent. You can repeat the same steps for other frames to get a full stack trace. If necessary you can also examine method arguments. More information about the stack layout can be found in Leandro Caniglia's document about process frames.

Of course it may be, that a crash happens at other locations. Things get tricky, if the crash happens inside an API call. The EBP chain may be broken or non obvious, because a lot of different information is being put on the stack. If you look at offset +1 (the return address) for each stack frame and put this information together, you will find similarities:

```
0013fd60 00b71c47
0013fd80 00b71be5
0013fd9c 00b92acf
0013fdb8 00bb527f
0013fdc8 00bb50cf
```

In our case these return addresses are pointing into the VM code cache. If you type the command 'LM' (list modules) in your debugger, you will see a listing of DLLs that are loaded in this process:

```
10000000 10048000   VVM31W
77f10000 77f59000   GDI32
7c800000 7c8f6000   kernel32
7e410000 7e4a1000   USER32
...
```

The return addresses found on the stack will give you a hint which module is responsible for which stack frame. If a crash should happen inside a Smalltalk primitive, the current EIP or any return addresses may point to the address range of the VM.

The debugger command 'K' (display stack trace) tries to walk the EBP chain and print the functions, modules for each stack frame. Of course this does not work for the dynamically generated JIT code, but it may display correct information for API calls (if you have the windows debug symbols installed).

Now, let's presume we did not know, why this crash happened.

Lets examine the current CPU instruction:

1001464c 8b78f8 mov edi,[eax-0x8] ds:0023:fffffffb=????????

What does it mean? This instruction tries to access memory based on the content of the EAX register. No lets examine the EAX register:

```
>r
eax=00000003 ebx=ffffe1f7 ecx=0000000a edx=10072a74 esi=00d687f4
edi=00000005
eip=1001464c esp=0013fd20 ebp=0013fd50 iopl=0         nv up ei pl
zr na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010247
```

EAX contains the integer 3. This is the encoded SmallInteger “1”.
SmallIntegers have a special encoding, because they have the LSB (least significant bit) set (1). All other objects pointers have this bit cleared (0). All objects besides SmallIntegers do have an object header, which is at offset -8. The CPU instruction that caused the crash, tried to access the object header to extract the objects size, flags or hash.

So what is the reason for this crash: the VM code searching the MDA for an implementation of a certain selector did not expect that the MDA does contain special objects like SmallIntegers.

That is what you usually see, when examining crashes: certain parts of the VM rely on a specific object layout. If this layout is broken, the VM may crash. These weaknesses/bugs are usually caused by a balance between speed and security. So the next step usually is: which code did break the object layout that is important for the VM.

Maybe this document will help you in such cases.