



# Using Memory Breakpoints with your Loaders

Shub-Nigurrath of ARTeam

Version 1.0 - April 2006

1.	Abstract .....	2
2.	Using Memory Breakpoints with OllyDbg .....	3
3.	Theory of Memory Breakpoints .....	3
3.1.	Paged Memory and access rights under Windows .....	3
3.2.	Creating Guard Pages .....	6
3.3.	What is the CPU Trap Flag and how to use it .....	6
3.4.	General structure of the Memory Breakpoints implementation .....	7
4.	A Sample code .....	8
4.1.	A sample (simple) victim .....	9
4.2.	The Loader of the sample victim using Memory BPs .....	9
5.	GetProcAddress a full a full example using Memory BPs .....	14
6.	References .....	16
7.	Conclusions .....	16
8.	History .....	17
9.	Greetings .....	17

## Keywords

Loader, memory breakpoints



### 1. Abstract

If you have ever used OllyDbg you surely have noticed the Memory Breakpoint feature which is really handy for packed applications, and generally for all those applications checking the presence of 0xCC bytes (normal breakpoints). It allows stopping the program when a memory location or range is accessed for execution, reading or writing. But how these breakpoints are created? They are not standard breakpoints supported by the CPU or directly by the Windows debug subsystem. Implementing it in your loaders is what was still missing; it requires a little more knowledge of the operating system.

This tutorial will discuss how memory breakpoints work and how to use them for you own loaders. It's an ideal prosecution of the already published Beginner's Tutorial #8 [1], where I already covered hardware and software breakpoints quite extensively (at beginner's level of course).

As usual I will provide sample code with this tutorial, and non-commercial sample victims. All the sources have been tested with Win2000/XP and Visual Studio 6.0.  
The techniques described here are general and not specific to any commercial applications. The whole document must be intended as a document on programming advanced techniques, how you will use these information will be totally up to your responsibility.

*Have Fun,  
Shub-Nigurrath*



## 2. Using Memory Breakpoints with OllyDbg

OllyDbg has an interesting alternative type of breakpoints, beside hardware and classical (0xCC based) breakpoints. These breakpoints had an interesting capability that is to not modify the memory under control, being so less detected, even less than hardware breakpoints. Hardware breakpoints can be easily detected through the CPU registers; normal breakpoints modify the memory and can be detected by self-checking applications. Memory breakpoints at the moment are much more difficult to be detected (for programmers lack of imagination mainly).

But how these breakpoints are working? It could be interesting to have them for our own debug loader..

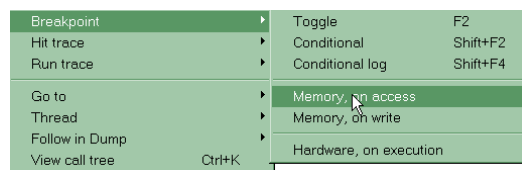


Figure 1 - OllyDbg contextual menu where we can set memory breakpoint

## 3. Theory of Memory Breakpoints

Before presenting the code used to implement memory breakpoints I need to present the required elements, for those of you who already doesn't know them (the other may skip this).

### 3.1. Paged Memory and access rights under Windows

Windows memory is a protected paginated memory. The discussion of what this means is a thing that I will assume as already known, otherwise the target of this tutorial would go too far. Anyway briefly each memory address is included into a container memory page and accessing to a specific memory address means to access to its container page. Windows memory access APIs are all making use of functions which access pages.

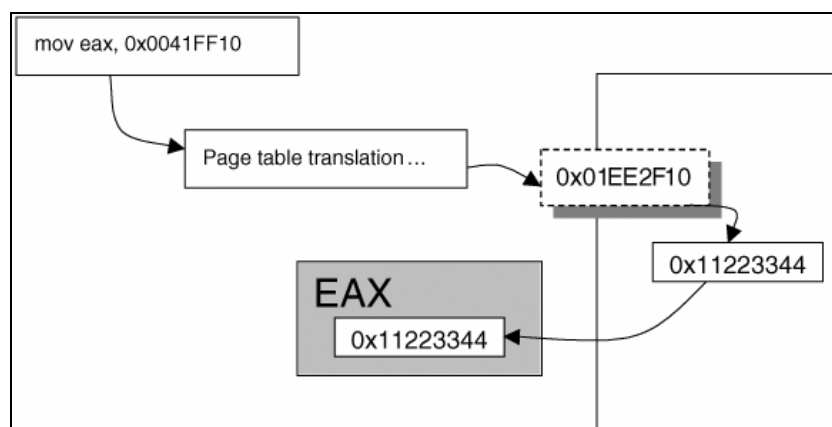


Figure 2 - Essential mechanism of Paged memory access

Figure 2 just recalls the paged mechanism for those of you not knowing it: when an instruction wants to recover a memory address, the system search the belonging page and then the memory address is recovered using the offset respect to the page start.



The access to these pages is protected by the means that each page has its own access settings, which specify read, write or execution rights for applications. This rather than being a security means is a functionality needed to organize the memory, which can store data as well as code, and how application will use it.

The main APIs used to access memory are WriteProcessMemory and ReadProcessMemory (there are other ways of course to access memory but these are the two most commonly used ones), already discussed in [2]. To change the pages rights there is another API, VirtualProtectEx (see Figure 3).

**VirtualProtectEx**  
The **VirtualProtectEx** function changes the protection on a region of committed pages in the virtual address space of a specified process.

```
BOOL VirtualProtectEx(  
    HANDLE hProcess,  
    LPVOID lpAddress,  
    SIZE_T dwSize,  
    DWORD flNewProtect,  
    PDWORD lpflOldProtect  
);
```

**Parameters**  
**hProcess**  
[in] Handle to the process whose memory protection is to be changed. The handle must have PROCESS\_VM\_OPERATION access. For more information on PROCESS\_VM\_OPERATION, see [OpenProcess](#).  
**lpAddress**  
[in] Pointer to the base address of the region of pages whose access protection attributes are to be changed. All pages in the specified region must be within the same reserved region allocated when calling the [VirtualAlloc](#) or [VirtualAllocEx](#) function using MEM\_RESERVE. The pages cannot span adjacent reserved regions that were allocated by separate calls to [VirtualAlloc](#) or [VirtualAllocEx](#) using MEM\_RESERVE.  
**dwSize**  
[in] Size of the region whose access protection attributes are changed, in bytes. The region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to (*lpAddress*+*dwSize*). This means that a 2-byte range straddling a page boundary causes the protection attributes of both pages to be changed.  
**flNewProtect**  
[in] Memory protection. This parameter can be one of the [memory protection constants](#). For pages that are already valid, this setting is ignored if it conflicts with the page's current setting specified using [VirtualAlloc](#) or [VirtualAllocEx](#).  
**lpflOldProtect**  
[out] Pointer to a variable that receives the previous access protection of the first page in the specified region of pages. If this parameter is NULL or does not point to a valid variable, the function fails.

**Return Values**  
If the function succeeds, the return value is nonzero.  
If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Figure 3 - What MSDN says about VirtualProtectEx

This API is an extension of VirtualProtect, which is support also operating on external processes. VirtualAlloc instead can work only on the calling process (indeed VirtualAlloc internally calls VirtualAllocEx passing its own PID).

There's a whole set of APIs used to handle pages rights among the Windows APIs, which are not much interesting for this tutorial (VirtualQueryEx, VirtualAllocEx, VirtualFreeEx, VirtualLock and VirtualUnlock, ...). Generally speaking the "Ex" suffix means that the API can operate on different processes (for a complete reference see [3]). Being our target to write a loader for an external program the "Ex" variants are mandatory for us.

Using the VirtualProtectEx is quite simple, as well as any other API of the same group. A typical call would look like:

```
DWORD dwNewProtect=PAGE_READWRITE;  
DWORD dwOldProtect=0;  
DWORD dwAddress=0x401000;  
VirtualProtectEx(pi.hProcess, (LPVOID)dwAddress, 1, dwNewProtect, &dwOldProtect);
```

for which I want to set to PAGE\_READWRITE the access rights of the page containing address 0x401000. Please note that the API will set the rights for the entire page even if the dwSize is set to 1 (see MSDN). The dwOldProtect will hold the previous access values. This will come handy writing loaders.

Below the currently supported flags, from MSDN:



## Using Memory Breakpoints with your Loaders

Value	Meaning
PAGE_EXECUTE 0x10	Enables execute access to the committed region of pages. An attempt to read or write to the committed region results in an access violation. This flag is not supported by <b>CreateFileMapping</b> .
PAGE_EXECUTE_READ 0x20	Enables execute and read access to the committed region of pages. An attempt to write to the committed region results in an access violation. This flag is not supported by <b>CreateFileMapping</b> .
PAGE_EXECUTE_READWRITE 0x40	Enables execute, read, and write access to the committed region of pages. This flag is not supported by <b>CreateFileMapping</b> .
PAGE_EXECUTE_WRITECOPY 0x80	Enables execute, read, and write access to the committed region of image file code pages. The pages are shared read-on-write and copy-on-write. This flag is not supported by <b>VirtualAlloc</b> , <b>VirtualAllocEx</b> , or <b>CreateFileMapping</b> .
PAGE_NOACCESS 0x01	Disables all access to the committed region of pages. An attempt to read from, write to, or execute the committed region results in an access violation exception, called a general protection (GP) fault. This flag is not supported by <b>CreateFileMapping</b> .
PAGE_READONLY 0x02	Enables read access to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only access and execute access, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE 0x04	Enables both read and write access to the committed region of pages.
PAGE_WRITECOPY 0x08	Gives copy-on-write protection to the committed region of pages. This flag is not supported by <b>VirtualAlloc</b> or <b>VirtualAllocEx</b> . <b>Windows Me/98/95:</b> This flag is not supported.
PAGE_GUARD 0x100	<p>Pages in the region become guard pages. Any attempt to access a guard page causes the system to raise a STATUS_GUARD_PAGE_VIOLATION exception and turn off the guard page status. Guard pages thus act as a one-time access alarm. For more information, see Creating Guard Pages. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over.</p> <p>If a guard page exception occurs during a system service, the service typically returns a failure status indicator.</p> <p>This value cannot be used with PAGE_NOACCESS.</p> <p>This flag is not supported by <b>CreateFileMapping</b>.</p> <p><b>Windows Me/98/95:</b> This flag is not supported. To simulate this behavior, use PAGE_NOACCESS.</p>
PAGE_NOCACHE 0x200	<p>Does not allow caching of the committed regions of pages in the CPU cache. The hardware attributes for the physical memory should be specified as "no cache." This is not recommended for general usage. It is useful for device drivers, for example, mapping a video frame buffer with no caching.</p> <p>This value cannot be used with PAGE_NOACCESS.</p> <p>This flag is not supported by <b>CreateFileMapping</b>.</p>
PAGE_WRITECOMBINE 0x400	<p>Enables write-combined memory accesses. When enabled, the processor caches memory write requests to optimize performance. Thus, if two requests are made to write to the same memory address, only the more recent write may occur.</p> <p>Note that the PAGE_GUARD and PAGE_NOCACHE flags cannot be specified with PAGE_WRITECOMBINE. If an attempt is made to do so, the SYSTEM_INVALID_PAGE_PROTECTION NT error code is returned by the function.</p> <p>This flag is not supported by <b>CreateFileMapping</b>.</p>



### 3.2. Creating Guard Pages

To describe what are the best way is to report what MSDN tells about.

A guard page provides a one-shot alarm for memory page access. This can be useful for an application that needs to monitor the growth of large dynamic data structures. For example, there are operating systems that use guard pages to implement automatic stack checking.

#### NOTE

Windows Me/98/95: You cannot create guard pages. To simulate this behaviour, use PAGE\_NOACCESS.

To create a guard page, set the PAGE\_GUARD page protection modifier for the page. This value can be specified, along with other page protection modifiers, in the VirtualAlloc, VirtualAllocEx, VirtualProtect, and VirtualProtectEx functions. The PAGE\_GUARD modifier can be used with any other page protection modifiers, except PAGE\_NOACCESS.

If a program attempts to access an address within a guard page, the system raises a STATUS\_GUARD\_PAGE\_VIOLATION (0x80000001) exception. The system also clears the PAGE\_GUARD modifier, removing the memory page's guard page status. The system will not stop the next attempt to access the memory page with a STATUS\_GUARD\_PAGE\_VIOLATION exception.

If a guard page exception occurs during a system service, the service fails and typically returns some failure status indicator. Since the system also removes the relevant memory page's guard page status, the next invocation of the same system service won't fail due to a STATUS\_GUARD\_PAGE\_VIOLATION exception (unless, of course, someone re-establishes the guard page).

Reading what I reported above from MSDN it come clear the first two blocks we require for writing code to handle memory breakpoints: VirtualProtectEx with PAGE\_GUARD and support for the properly raised exception STATUS\_GUARD\_PAGE\_VIOLATION in our debug loader (see [2]).

### 3.3. What is the CPU Trap Flag and how to use it

Having the above elements is anyway not enough for us. We need another last element to combine. The x86 Intel CPUs have a special flag called TS (Trap Flag) which the user can set and reset. The meaning of this flag is to instruct the CPU to go in single step mode, executing each ASM instruction and then stopping a the beginning of the following op-code. The system also raises an exception EXCEPTION\_SINGLE\_STEP.

The TF is the 8<sup>th</sup> bit of the EFLAGS register and is used to enable single-step mode for debugging; clear to disable single-step mode. Intel's documentation reports “..should not be modified by application programs...”.. of course we will! 😊



A simple function useful to set and unset this flag is

```
//If bSet==1 then set the single step, otherwise unset it.
BOOL SetSingleStepMode(HANDLE hThread, BOOL bSet) {
    CONTEXT context;
    context.ContextFlags=CONTEXT_FULL;
    GetThreadContext(hThread,&context);
    if(bSet)
        context.EFlags |= 0x00000100;
    else
        context.EFlags &= ~0x00000100;
    return SetThreadContext(hThread, &context);
}
```

Now we have all the elements required to write our own memory breakpoint handing code.

### **3.4. General structure of the Memory Breakpoints implementation**

Before going further with the code I must say that writing a loader which only set a Page Guard without taking care of the Trap Flag will not advance. Suppose this situation:

1. We wrote a debug loader handing the Page Guard Exception (EIP points to the program's EP).
2. The loader sets the Page Guard at the memory address where we want to stop.
3. The program starts and the loader enters the debug cycle.
4. We receive a Page Guard Exception. According to documentation the Page Guard should now be removed and then we should be able to set it again in order to stop again a next time.
5. We can check EIP and see if it matches the memory address where we wanted to stop. In case we exit otherwise we set again the Page Guard and continue with debugging.

The problem with this sequence is that EIP never gets a chance to advance and move from the EP, so we will always stay at the EP and never gets move forward. We are in a dead-lock.

What we need is a different solution that involves the Single Step mode.

To fully explain the structure of the code I drawn the structure of the program as reported by Figure 4.

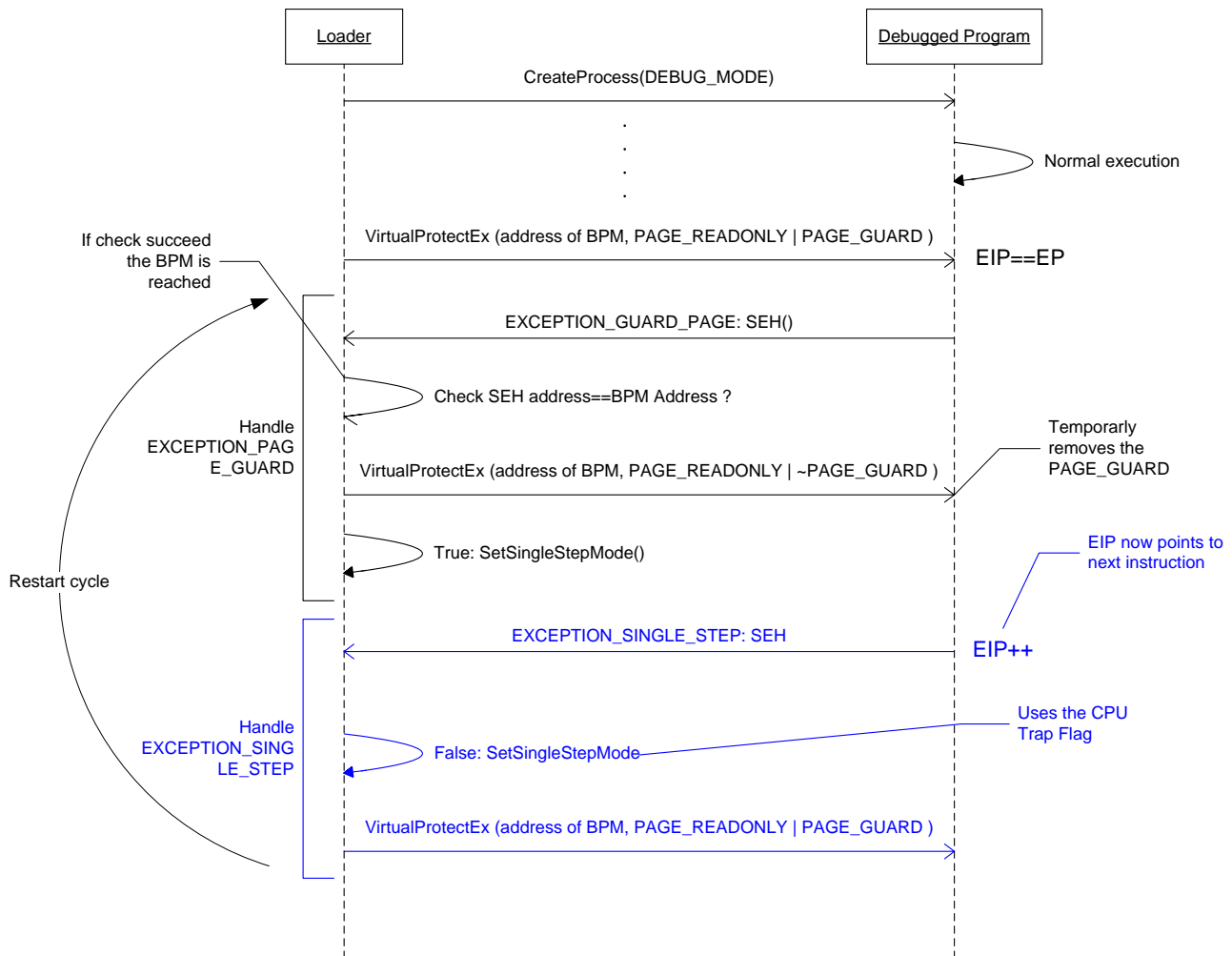


Figure 4 - General Sequence Diagram of a Loader implementing a Memory BP and its interactions with Target

Figure 4 is quite explanatory; anyway the concept is in essence simple: we bounce between `EXCEPTION_PAGE_GUARD` and `EXCEPTION_SINGLE_STEP` setting and unsetting Page Guards. Each time we will check if the memory address where the Page Guard Exception happened matches the memory address of the Memory Breakpoint.

The EIP has been incremented of course before the Single Step Exception is raised.

## 4. A Sample code

Starting from the already known (see [2]) debug cycle I wrote this stripped down code, containing only the essential things. As usual I wrote a simple non commercial victim, just to avoid problems, but you are free to use this code with whatever program you like 🤖





### 4.1. A sample (simple) victim

The victim is simple (extremely), it's only a program for which you have to break before a specific nag call (see Figure 5). This is enough for us. The victim is simple; the task it asks you to do is not instead.

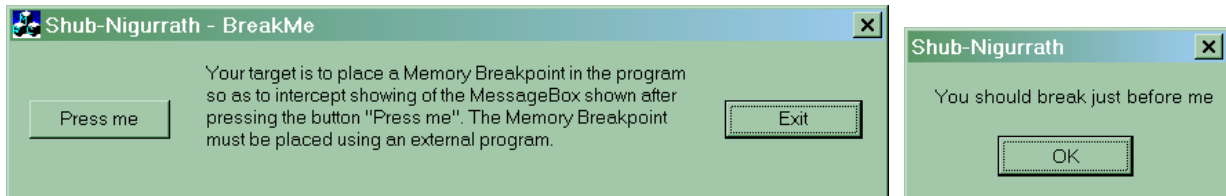


Figure 5 - Breakme used for Example 1

The address where we want to place a memory breakpoint is 0x401490, as reported in Figure 6.

00401480	. 8B41 20	MOV EAX,DWORD PTR DS:[ECX+20]	
00401483	. 6A 00	PUSH 0	
00401485	. 68 40304000	PUSH BreakMe.00403040	
0040148A	. 68 20304000	PUSH BreakMe.00403020	
0040148F	. 50	PUSH EAX	
00401490	. FF15 CC214000	CALL DWORD PTR DS:[&USER32.MessageBoxA	Style = MB_OK MB_APPLMODAL Title = "Shub-Nigurrath" Text = "You should break just before me" hOwner = NULL
00401496	. C3	RETN	

Figure 6 - Where to place a Memory BP on Breakme

#### NOTE

The code of this program is under the example\_1\ folder

### 4.2. The Loader of the sample victim using Memory BPs

We have now a target and we know what to do, have also the instruments then we are ready to write the program. I report here the whole example1's code. See just after it for comments

```
<----- Start Code Snippet ----->

// Shub-Nigurrath memory breakpoint simulator

//Note that all the includes are into stdafx.h, just to conform to MFC standard.
#include "stdafx.h"

#define _countof(array) \
    (sizeof(array)/sizeof(array[0]))

//An useful define to allow correct default handling for each exception. This is the first thing to call inside
any case of the debug switch.
//In the case the dwFirstChance is not true means that process is as good as dead, so it's useless to
//continue with our debugger.
//Handler of the debug SEH must, if needed change the value of dwContinueStatus
#define FIRSTCHANCEHANDLING() \
    if (DebugEv.u.Exception.dwFirstChance) { \
        contproc = TRUE; \
        dwContinueStatus = DBG_EXCEPTION_NOT_HANDLED; \
    } \
    else { \
        contproc = FALSE; \
    }

BOOL SetSingleStepMode(HANDLE hThread, BOOL bSet);

int main(int argc, char* argv[])
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    char FileName[MAX_PATH];
```





## Using Memory Breakpoints with your Loaders

```
BOOL contproc=TRUE;
DWORD    dwContinueStatus=DBG_CONTINUE;

HANDLE    hSaveFile=0;
HANDLE    hSaveProcess=0;
HANDLE    hSaveThread=0;
LPVOID    lpMsgBuf=NULL;
DWORD    dwRead=0;

DEBUG_EVENT DebugEv;    // debugging event information
DWORD dwOldProtect=-1;  //old protection's page value

#ifdef _DEBUG
strcpy(FileName, "..\\bin\\BreakMe.exe");
#else
strcpy(FileName, ".\\BreakMe.exe");
#endif

printf("\n+-----+\n");

//Some parameters we will use later for the VirtualProtectEx
DWORD dwGuardProtect=PAGE_READONLY | PAGE_GUARD;

//Address of where we want to place the Memory Breakpoint!
//00401490 . FF15 CC214000 CALL DWORD PTR DS:[<&USER32.Message>; \MessageBoxA
DWORD dwAddress=0x00401490;

ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );

printf("-> Started tracing\n");

// Start the child process.
if(!CreateProcess( NULL, // No module name (use command line).
    FileName,          // Command line.
    NULL,              // Process handle not inheritable.
    NULL,              // Thread handle not inheritable.
    TRUE,              // Set handle inheritance.
    DEBUG_PROCESS,     // No creation flags (use for more than 1 process
    NULL,              // Use parent's environment block.
    NULL,              // Use parent's starting directory.
    &si,               // Pointer to STARTUPINFO structure.
    &pi ))             // Pointer to PROCESS_INFORMATION structure.
{
    printf("Create Process Failed: Unexpected load error\n");
    return 1;
}

//Read the initial value of the memory page containing the address on which we want to place
//our memory breakpoint.
VirtualProtectEx(pi.hProcess, (LPVOID)dwAddress, 1, dwGuardProtect, &dwOldProtect);

while (TRUE) {
    // Wait for a debugging event to occur. The second parameter indicates
    // that the function does not return until a debugging event occurs.

    if (WaitForDebugEvent(&DebugEv, 1000)) { // wait 1 second
        // Process the debugging event code.

        switch (DebugEv.dwDebugEventCode) {

            case EXCEPTION_DEBUG_EVENT: {
                // Process the exception code. When handling exceptions, remember
                // to set the continuation status parameter (dwContinueStatus).
                // This value is used by the ContinueDebugEvent function.

                switch (DebugEv.u.Exception.ExceptionRecord.ExceptionCode) {

                    case EXCEPTION_GUARD_PAGE: {
                        FIRSTCHANCEHANDLING();

                        printf("Guard Page Hit -> Exception address:%08X\n",
                            DebugEv.u.Exception.ExceptionRecord.ExceptionAddress);
                        contproc = TRUE;

                        #ifdef _DEBUG
                        CONTEXT context;
                        context.ContextFlags=CONTEXT_FULL;
                        GetThreadContext(pi.hThread,&context);
                        assert((DWORD)(context.Eip) ==
                            (DWORD)(DebugEv.u.Exception.ExceptionRecord.ExceptionAddress));
                        #endif

                        VirtualProtectEx(pi.hProcess, (LPVOID)dwAddress, 1, dwGuardProtect, NULL);
```





```
        );
    }
    break;

    default: {
        FIRSTCHANCEHANDLING();

        contproc = TRUE;
        dwContinueStatus = DBG_CONTINUE;
    }
    break;

} //end switch DebugEv.dwDebugEventCode

if (!contproc)
    break;

ContinueDebugEvent(DebugEv.dwProcessId, DebugEv.dwThreadId, dwContinueStatus);

} //end if

} //end while

// Close process and thread handles.
CloseHandle( pi.hProcess );
CloseHandle( pi.hThread );

//Detach from debugger/debuggee
printf("Process Id=%X\n Terminated!\n", pi.dwProcessId);

DebugActiveProcessStop(pi.dwProcessId);

printf("+-----+\n");

ExitProcess(0);
return 0;
}

//If bSet==1 then set the single step, otherwise unset it.
BOOL SetSingleStepMode(HANDLE hThread, BOOL bSet) {
    CONTEXT context;
    context.ContextFlags=CONTEXT_FULL;
    GetThreadContext(hThread,&context);
    if(bSet)
        context.EFlags |= 0x00000100;
    else
        context.EFlags &= ~0x00000100;
    return SetThreadContext(hThread, &context);
}

<----- End Code Snippet ----->
```

8

Here the important points (refer also to Figure 4):

1. This is just a macro I used to let the code in the big debugging switch be more readable. It just include what is also reported in the Matt Pietrek's Paper [4], tied to the way Windows handle exceptions: when the exception is received by the debugger with dwFirstChance set to TRUE the program has just raised the exception and has still not gone through the default exception mechanism (the system), so is the right place where to try to recover it (that is what we do indeed), resuming it's execution. When instead dwFirstChance is FALSE, the program has already entered the SEH system mechanism and this time is the last chance the system gives to the debugger to close all gently. The process is now as good as dead and we close the loader.
2. Details about the memory breakpoint we want to set and the page guard protections we use.
3. First call to VirtualProtectEx where we set the Page Guard for the first time.
4. The EXCEPTION\_GUARD\_PAGE is received and then we have here to check if the Exception address
5. The ExceptionAddress is equal to the EIP where the exception has been raised, than the instruction we were trying to execute. If it is equal to the address we specified at point 2



- then we are happy and we consider our task closed. If not we remove the Page Guard and set the single step mode.
6. The EXCEPTION\_SINGLE\_STEP is received. At this point we can remove the single step mode and restore the Page Guard instead.
  7. Point 7 is useless for our example but I left it here because gives some useful information about the debugged program might help to understand what's happening.
  8. SetSingleStepMode is a simple function (presented earlier) which set and unset the T-Flag.

The program once run reports these results: when you press the “Press me” button a new dialog now appears<sup>1</sup>:

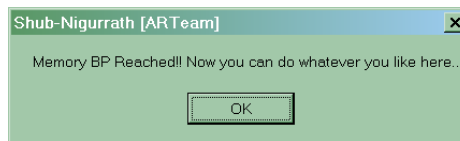


Figure 7 - Memory BP successfully reached

The output from the DOS window is something like the following:

```
<----- Start Output Snippet ----->

-> Started tracing
CREATE_PROCESS_DEBUG_EVENT
    hFile:7B8
    ProcessId:14A8
    hProcess:7D4
    hThread:7D0
    lpBaseOfImage:00400000
    dwDebugInfoFileOffset:0
    nDebugInfoSize:0
    lpThreadLocalBase:7FFDF000
    lpStartAddress:00401700
    lpImageName:00000000
    fUnicode:1
This computer has page size 1000.
+-----+
Guard Page Hit -> Exception address:00401700
Single Step -> Exception address:00401701
Guard Page Hit -> Exception address:00401701
Single Step -> Exception address:00401703
Guard Page Hit -> Exception address:00401703
Single Step -> Exception address:00401705
Guard Page Hit -> Exception address:00401705
Single Step -> Exception address:0040170A
Guard Page Hit -> Exception address:0040170A
Single Step -> Exception address:0040170F
...
Single Step -> Exception address:00401490
Guard Page Hit -> Exception address:00401490
Memory BP Reached!
Single Step -> Exception address:77D804EA
...

<----- End Output Snippet ----->
```

It is clearly visible that the EIP advances step by step of a number of bytes equal to the op-code dimension of the just executed ASM instruction.

<sup>1</sup> Pay attention that usually it is in background.



### NOTE

Of course you can code your own memory breakpoint using an alternative language and more compact too, if you do not have educational intents. This code is specifically written to be explained and understood.

## 5. GetProcTracer a full a full example using Memory BPs

We are then ready to code something more useful. What I decided to code as a bonus to this tutorial is a GetProcAddress calls tracer. The problem is quite urgent because almost all packers around started to destroy the IAT of compressed programs because they finally realized that it can be used to easily rebuild unpacked programs 🤖. To avoid using the IAT they have several methods but one of the most used APIs is always the GetProcAddress, eventually masked inside the unpacking code<sup>2</sup>. The API, as you know is used to retrieve the entry point of specified APIs given the handle to a library.

### NOTE

The code of this program is under the GetProcTracer\ folder

What this tracer does is then to place a memory breakpoint in a proper point of the GetProcAddress API and return to the user (and to a log file) the parameters used by the caller of this API. The code is a little more complex than the example above, so I decided to leave it out of this document, but you can find the whole sources into this same archive. See these sources for details and comments.

The syntax is in Figure 8.

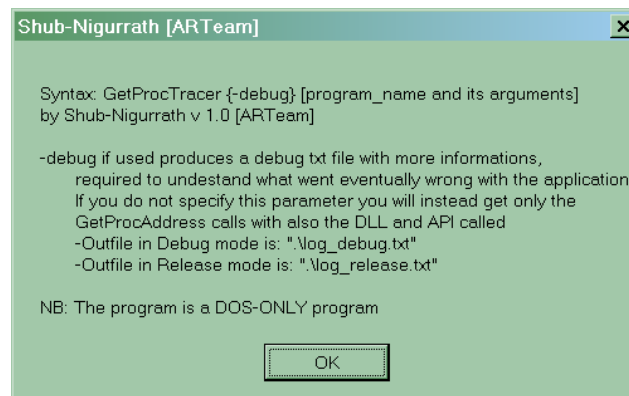


Figure 8 - GetProcTracer command-line syntax

The “-debug” switch is indeed giving back a lot of information, so better using just to understand what is not working for a specific program, normally it should be left out.

<sup>2</sup> Obviously this is not true for all cases, other packer avoid calling this API using hashes of the APIs to be called and directly accessing to the right locations, or using a private custom version equivalent to GetProcAddress.



We can try this tracer on the previous breakme using the following command-line

```
GetProcTracer.exe ..\..\example1\bin\breakme.exe
```

The resulting log\_release.txt is quite long, but what follows is a snippet:

```
<----- Start Output Snippet ----->

[0 - 09:59:32,258] Parameters used for this Tracing..
    API2HOOK=LdrGetProcedureAddress
    LIBRARY2HOOK=NTDLL.DLL
    API2HOOK_OFFSET=0x13
[1 - 09:59:35,974] GetProcAddress (LoadLibrary("kernel32.dll"), "InitializeCriticalSectionAndSpinCount")
[2 - 09:59:36,052] GetProcAddress (LoadLibrary("kernel32.dll"), "ProcessIdToSessionId")
[3 - 09:59:36,115] GetProcAddress (LoadLibrary("USER32.dll"), "GetSystemMetrics")
[4 - 09:59:36,177] GetProcAddress (LoadLibrary("USER32.dll"), "MonitorFromWindow")
[5 - 09:59:36,224] GetProcAddress (LoadLibrary("USER32.dll"), "MonitorFromRect")
[6 - 09:59:36,271] GetProcAddress (LoadLibrary("USER32.dll"), "MonitorFromPoint")
[7 - 09:59:36,318] GetProcAddress (LoadLibrary("USER32.dll"), "EnumDisplayMonitors")
[8 - 09:59:36,380] GetProcAddress (LoadLibrary("USER32.dll"), "EnumDisplayDevicesW")
[9 - 09:59:36,427] GetProcAddress (LoadLibrary("USER32.dll"), "GetMonitorInfoW")
[10 - 09:59:36,489] GetProcAddress (LoadLibrary("COMCTL32.DLL"), "InitCommonControlsEx")
[11 - 09:59:36,536] GetProcAddress (LoadLibrary("COMCTL32.DLL"), "InitCommonControlsEx")
[12 - 09:59:36,568] GetProcAddress (LoadLibrary("COMCTL32.DLL"), "InitCommonControlsEx")
[13 - 09:59:36,614] GetProcAddress (LoadLibrary("COMCTL32.DLL"), "InitCommonControlsEx")
[14 - 09:59:36,661] GetProcAddress (LoadLibrary("COMCTL32.DLL"), "InitCommonControlsEx")
[15 - 09:59:36,708] GetProcAddress (LoadLibrary("COMCTL32.DLL"), "InitCommonControlsEx")
[16 - 09:59:38,363] GetProcAddress (LoadLibrary("ntdll.dll"), "NtQueryInformationProcess")
[17 - 09:59:38,457] GetProcAddress (LoadLibrary("breakme.exe"), "CtfImmCoUninitialize")
[18 - 09:59:38,457] GetProcAddress (LoadLibrary("breakme.exe"), "CtfImmLastEnabledWndDestroy")
[19 - 09:59:38,473] GetProcAddress (LoadLibrary("breakme.exe"), "CtfImmSetCiceroStartInThread")
[20 - 09:59:38,473] GetProcAddress (LoadLibrary("breakme.exe"), "CtfImmIsCiceroStartedInThread")
[21 - 09:59:38,488] GetProcAddress (LoadLibrary("breakme.exe"), "CtfImmIsCiceroEnabled")
...

<----- End Output Snippet ----->
```

As you can see the program is able to rebuild the original call to GetProcAddress, and the library to which the handle owns. The reference to the debugged program itself is tied to the loading Windows loading mechanism. The results got from this tracer are quite interesting indeed, but you can analyze them on your own.

I found 3 possible hooking points which I implemented into the program as defines (I'm lazy you have to recompile if you want to enable one of them). These define are:

```
<----- Start Code Snippet ----->

////////////////////////////////////
//Hook details!
//define API2HOOK "GetProcAddress"
// // #define API2HOOK_EP
// #define LIBRARY2HOOK "KERNEL32.DLL"
// #define API2HOOK_OFFSET 0xC
//DWORD OridataAtOffset[3] = { 0xBB, 0xFF, 0xFF}; // MOV EBX, 0xFF

//Works and is not detected by AsProtect
#define API2HOOK "LdrGetProcedureAddress"
// #define API2HOOK_EP 0x7c919b88 // If LoadLibrary fails this address is used, if defined. Useful when
// the API is not in the Export Table of the DLL
#define LIBRARY2HOOK "NTDLL.DLL"
#define API2HOOK_OFFSET 0x13
DWORD OridataAtOffset[4] = { 0xE8, 0x15, 0xFE, 0xFF };

// #define API2HOOK "GetProcAddress"
// // #define API2HOOK_EP
// #define LIBRARY2HOOK "KERNEL32.DLL"
// #define API2HOOK_OFFSET 0x39
//DWORD OridataAtOffset[3] = { 0xE8, 0xB7, 0xFF };

<----- End Code Snippet ----->
```

If you enable one of them the program will change the behaviour and also the output.

- The API2HOOK\_EP deserves an extra explanation. Think for example to be trying to place a memory breakpoint at a specific offset of an internal API. This operation requires first of





all finding the entry point of this function, but if the function is not in the export table of the Dll you will not be able to use LoadLibrary; with these cases an address is then used. I placed this address into a separate define, because might change between different system upgrades (for example between XP SP1 and XP SP2). The specific example of LdrGetProcedureAddress doesn't need this attention: it is exported as a forwarder by the Kernel32.dll to the real export into NTDll.dll.

- LIBRARY2HOOK is the library containing the API to hook (either exported or not)
- API2HOOK\_OFFSET is the offset from the API entry point where we want to place a memory breakpoint.

The compiled version is using LdrGetProcedureAddress because is not usually checked by AsProtect and other packers.

The overall GetProcTracer sources are meant to be the base for other customizations you might want to do, I wrote them expressly clear, with comments and redundant.

## 6. References

- [1] "Beginner tutorial #8, Breakpoints Theory", Shub-Nigurrath of ARTeam, <http://tutorials.accessroot.com>
- [2] "Cracking with Loaders: Theory, General Approach and a Framework, Version 1.2", Shub-Nigurrath of ARTeam, <http://tutorials.accessroot.com>
- [3] "Platform SDK: Memory Management Functions", MSDN Library, [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/memory\\_management\\_functions.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/memory_management_functions.asp)
- [4] "An In-Depth Look into the Win32 Portable Executable File Format, Part 1 and 2", Matt Pietrek, <http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx>
- [5] "Guide on How to play with processes memory, write loaders and Oraculums", Shub-Nigurrath of ARTeam, <http://tutorials.accessroot.com>
- [6] "Writing Loaders for Dlls: theory and techniques Version 1.0", Shub-Nigurrath of ARTeam, <http://tutorials.accessroot.com>

## 7. Conclusions

Well, this is the end of this story, the memory breakpoints can be used in different interesting ways. I hope to have explained what they are and how to use for your own code. Of course, the code here reported is long and quite big for a thing that could be coded in ASM with few lines, but I expressly coded these examples to be didactical and self-explanatory. The essence of memory breakpoints is simple after all.

Changing the test condition of the memory breakpoint you can also have a program testing the access to specific memory regions (for example the .code section) to build automatic unpacker or intelligent debugger loader. Of course this is done at the price of increased target slowness. The real solution is a mixture of techniques as you do when you normally use OllyDbg.

Generally speaking it is quite simple to write a complete debugger loader mixing the techniques already described in [2, 5] with memory breakpoints. The features you will have will be: memory search patterns (discussed in [5]), hardware and software breakpoints (introduced in [2] and explained by the code point of view in [6]), memory breakpoints (here), and anti-anti-debugging





(discussed in [2]). What you will easily get will be a loader powerful as much as OllyDbg (well not exactly but at least you can think of).

**All the code provided with this tutorial is free for public use, just make a greetz to the authors and the ARTeam if you find it useful to use. Don't use these concepts for making illegal operation, all the info here reported are only meant for studying and to help having a better knowledge of application code security techniques.**

## 8. History

- Version 1.0 – First public release!

## 9. Greetings

I wish to tank all the ARTeam members of course and who read the beta versions of this tutorial and contributed. Thanks deroko too for his hints .... and of course you, who are still alive at the end of this quite document!



<http://cracking.accessroot.com>