Using Memory Breakpoints with your Loaders

Shub-Nigurrath of ARTeam

Version 1.0 – April 2006

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 Phun,
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..

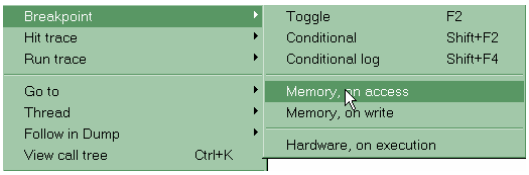

Figure 1 - OllyDbg contextual menu where we can set memory breakpoint

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.

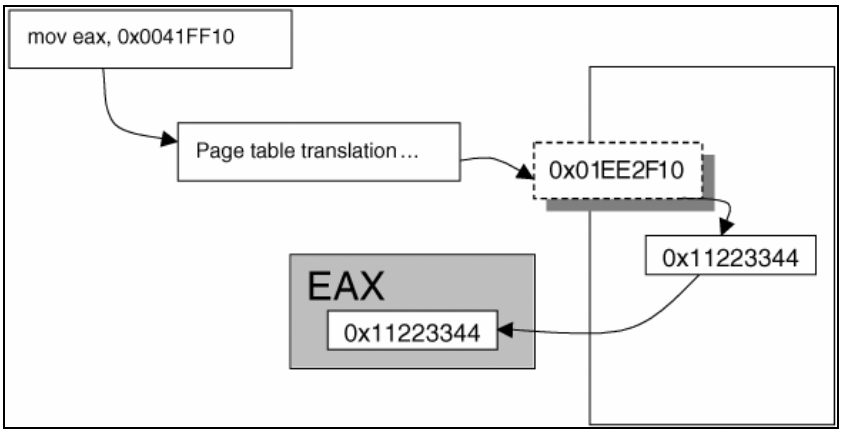

Figure 2 - Essential mechanism of Paged memory access

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, witch 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_READWITE 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:

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 **CreateFileMapping**.
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 **CreateFileMapping**.
PAGE_EXECUTE_READWRITE 0x40	Enables execute, read, and write access to the committed region of pages. This flag is not supported by **CreateFileMapping**.
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 **VirtualAlloc**, **VirtualAllocEx**, or **CreateFileMapping**.
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 **CreateFileMapping**.
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 **VirtualAlloc** or **VirtualAllocEx**. **Windows Me/98/95:** This flag is not supported.
PAGE_GUARD 0x100	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. If a guard page exception occurs during a system service, the service typically returns a failure status indicator. This value cannot be used with PAGE_NOACCESS. This flag is not supported by **CreateFileMapping**. **Windows Me/98/95:** This flag is not supported. To simulate this behavior, use PAGE_NOACCESS.
PAGE_NOCACHE 0x200	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. This value cannot be used with PAGE_NOACCESS. This flag is not supported by **CreateFileMapping**.
PAGE_WRITECOMBINE 0x400	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. 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. This flag is not supported by **CreateFileMapping**

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 8th 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! 😉