A quick follow-up to the previous post on hardware breakpoints. One thing that people consider when using hardware breakpoints is that they're only limited to four addresses to break on within a thread. This is true, and something that cannot be overcome. There are some interesting techniques like executing instructions in one thread through a different one with breakpoints on it, but something like that is rarely done and hardly worth the effort to code correctly. The next viable alternative is done through memory breakpoints. This involves marking the page that the desired address is on as a guard page. As the MSDN documentation says, once a page is marked as a guard page, any access to it raises a `STATUS_GUARD_PAGE_VIOLATION` exception. The guard flag is then removed and any further accesses are allowed, assuming the guard flag has not been set again. The technique then is simple — find the page that the desired address is on, set it as a guard page, and catch the exception. There is a slight nuance however. Since access to any instruction on the page triggers the `STATUS_GUARD_PAGE_VIOLATION` exception, trying to reset the guard page status and continue execution in the handler will just result in a loop. A jump to a stub function cannot be used since it is not known what address raised the exception. Thus, the EIP also cannot be safely modified in the `STATUS_GUARD_PAGE_VIOLATION` handler. What is required is setting the trap flag so the EIP increments before the `EXCEPTION_SINGLE_STEP` is raised. This `EXCEPTION_SINGLE_STEP` is also caught and that is where everything takes place. Just like hardware breakpoints, all that is needed is to check if `ExceptionAddress` is equal to the desired address. If so, then you've got the context of the thread and can do as you please. The important thing is to set the page back to a guard page prior to leaving the handler. How the exception handler and how the breakpoint is added is shown below. The code is written for the test application for the hardware breakpoints article and is attached below.

Exception handler:

```
LONG WINAPI ExceptionFilter(PEXCEPTION_POINTERS ExceptionInfo) {
    if(ExceptionInfo->ExceptionRecord->ExceptionCode ==
STATUS_GUARD_PAGE_VIOLATION) {
        ExceptionInfo->ContextRecord->EFlags |= 0x100;
                return EXCEPTION_CONTINUE_EXECUTION;
    }
    else if(ExceptionInfo->ExceptionRecord->ExceptionCode ==
EXCEPTION_SINGLE_STEP) {
        if((DWORD)ExceptionInfo->ExceptionRecord->ExceptionAddress ==
(DWORD)func_addr) {
            PCONTEXT debug_context = ExceptionInfo->ContextRecord;
            printf("Breakpoint hit!\n");
            print_parameters(debug_context);
            modify_text(debug_context);
        }
        MEMORY_BASIC_INFORMATION mem_info;
        DWORD old_protections = 0;
        VirtualQuery((LPCVOID)func_addr, &mem_info,
sizeof(MEMORY_BASIC_INFORMATION));
        VirtualProtect(mem_info.BaseAddress, mem_info.RegionSize,
mem_info.AllocationProtect | PAGE_GUARD,
            &old_protections);
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}
```

Setting the breakpoint:

```
        DWORD old_protections = 0;
        VirtualQuery((LPCVOID)func_addr, &mem_info,
sizeof(MEMORY_BASIC_INFORMATION));
        VirtualProtect(mem_info.BaseAddress, mem_info.RegionSize,
mem_info.AllocationProtect | PAGE_GUARD,
            &old_protections);
```

It is a pretty interesting technique, and is also how OllyDbg implements its memory breakpoints. The downside though is that it works at the page boundary and can considerably slow down execution time — depending on how often the page is accessed.
Full source for memory breakpoints can be found here
A downloadable PDF of this post can be found here