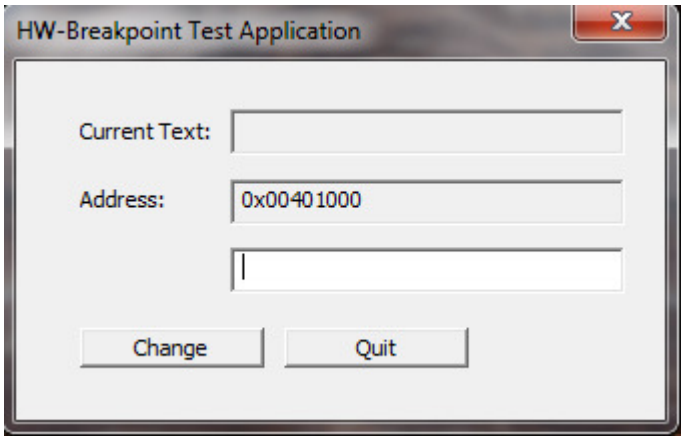
Some good references to read prior to this post. In short, to use hardware breakpoints there are eight debug registers (DR0 to DR7) that can be utilized. Eight, however, is a bit of an overstatement — DR4 and DR5 are no longer used and their functionality is replaced with DR6 and DR7, so there are really six. The debug registers DR0 – DR3 can each hold a linear address to break on depending on how the debug control (DR7) register is set. The debug status (DR6) register lets a debugger determine which debug conditions have occurred. Therefore, you are permitted four addresses to set hardware breakpoints on (assuming that they're not being chained across threads). What is the utility of these breakpoints? For one, they don't modify the code that breakpoints are hit on. This is especially useful in the context of defeating simple anti-debugging mechanisms that check function prologues for hooks. All that is required is to install a run-time exception handler and set up hardware breakpoints in the context of the applications main thread, or the thread that is executing the desired code to break on. This can be done with Windows Structured Exception Handling capabilities. Structured Exception Handlers (SEHs) in Windows are stored as a linked list. When an exception is raised, this list is traversed until a handler for the exception is found. If one is found then the handler gains execution of the program and handles the exception. If one is not found then the application goes into an undefined state and may crash depending on the type of exception. Vectored Exception Handler (VEHs) are extensions of SEH that can be installed to watch and handle all exceptions that an application generates. In an application, VEHs are typically added through `AddVectoredExceptionHandler` instead of `__try/__except` blocks like SEH. This, however, is pretty irrelevant in terms of hack development — both SEHs and VEHs should be added at runtime. The main benefit that VEHs have is that they are always invoked prior to SEHs (being a "new" feature included in WinXP), and that `AddVectoredExceptionHandler` lets you specify whether you want your exception handler to be the first one to be called when an exception is raised. This leads most people to prefer VEHs over SEHs nowadays.

To illustrate SEH/VEH I've developed a sample application. An injected DLL will install SEH and VEH handlers that will break upon entry to a certain function (whose address is noted in the dialog field for convenience). The goal is to break on the function that takes the text in the enabled edit control and puts it in the disabled edit control following "Current Text:".



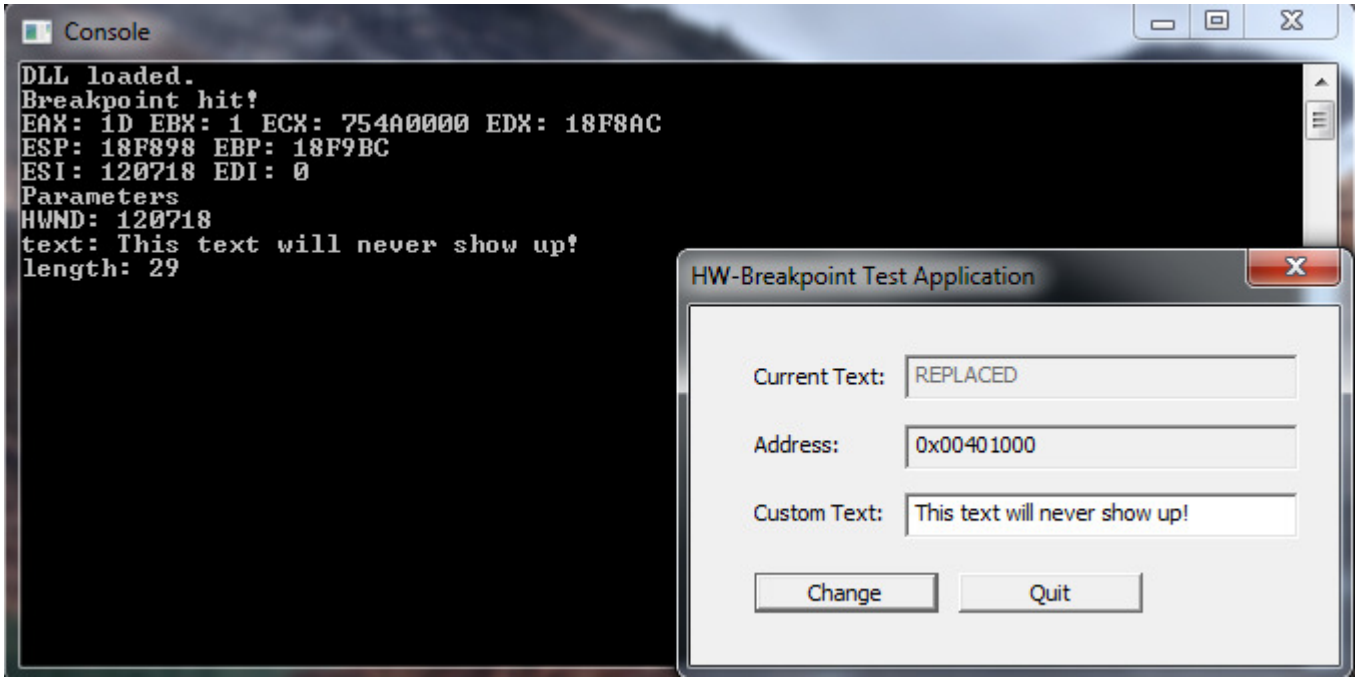
The code for all of this is relatively straightforward. The full listing using SEH is shown below:

```
#include <Windows.h>
```

```
#include <TlHelp32.h>
#include <stdio.h>

const DWORD func_addr = 0x00401000;
const DWORD func_addr_offset = func_addr + 0x1;

void print_parameters(PCONTEXT debug_context) {
    printf("EAX: %X EBX: %X ECX: %X EDX: %X\n",
        debug_context->Eax, debug_context->Ebx, debug_context->Ecx,
debug_context->Edx);
    printf("ESP: %X EBP: %X\n",
        debug_context->Esp, debug_context->Ebp);
    printf("ESI: %X EDI: %X\n",
        debug_context->Esi, debug_context->Edi);
    printf("Parameters\n"
        "HWND: %X\n"
        "text: %s\n"
        "length: %i\n",
        (HWND)(*(DWORD*)(debug_context->Esp + 0x4)),
        (char*)(*(DWORD*)(debug_context->Esp + 0x8)),
        (int)(*(DWORD*)(debug_context->Esp + 0xC)));

}

void modify_text(PCONTEXT debug_context) {
    char* text = (char*)(*(DWORD*)(debug_context->Esp + 0x8));
    int length = strlen(text);
    _snprintf(text, length, "REPLACED");
}

void __declspec(naked) change_text_stub(void) {
    __asm {
        push ebp
        jmp [func_addr_offset]
    }
}

LONG WINAPI ExceptionFilter(PEXCEPTION_POINTERS ExceptionInfo) {
    if(ExceptionInfo->ExceptionRecord->ExceptionCode ==
EXCEPTION_SINGLE_STEP) {
        if((DWORD)ExceptionInfo->ExceptionRecord->ExceptionAddress ==
func_addr) {
            PCONTEXT debug_context = ExceptionInfo->ContextRecord;
            printf("Breakpoint hit!\n");
            print_parameters(debug_context);
            modify_text(debug_context);
            debug_context->Eip = (DWORD)&change_text_stub;
            return EXCEPTION_CONTINUE_EXECUTION;
        }
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

void set_breakpoints(void) {
    HANDLE hTool32 = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if(hTool32 != INVALID_HANDLE_VALUE) {
        THREADENTRY32 thread_entry32;
```

```
        thread_entry32.dwSize = sizeof(THREADENTRY32);
        FILETIME exit_time, kernel_time, user_time;
        FILETIME creation_time;
        FILETIME prev_creation_time;
        prev_creation_time.dwLowDateTime = 0xFFFFFFFF;
        prev_creation_time.dwHighDateTime = INT_MAX;
        HANDLE hMainThread = NULL;
        if(Thread32First(hTool32, &thread_entry32)) {
            do {
                if(thread_entry32.dwSize >= FIELD_OFFSET(THREADENTRY32,
th32OwnerProcessID) + sizeof(thread_entry32.th32OwnerProcessID)
                    && thread_entry32.th32OwnerProcessID ==
GetCurrentProcessId()
                    && thread_entry32.th32ThreadID != GetCurrentThreadId()) {
                        HANDLE hThread = OpenThread(THREAD_SET_CONTEXT |
THREAD_GET_CONTEXT | THREAD_QUERY_INFORMATION,
                            FALSE, thread_entry32.th32ThreadID);
                        GetThreadTimes(hThread, &creation_time, &exit_time,
&kernel_time, &user_time);
                        if(CompareFileTime(&creation_time,
&prev_creation_time) == -1) {
                            memcpy(&prev_creation_time, &creation_time,
sizeof(FILETIME));
                            if(hMainThread != NULL)
                                CloseHandle(hMainThread);
                            hMainThread = hThread;
                        }
                        else
                            CloseHandle(hThread);
                }
                thread_entry32.dwSize = sizeof(THREADENTRY32);
            } while(Thread32Next(hTool32, &thread_entry32));
            (void)SetUnhandledExceptionFilter(ExceptionFilter);
            CONTEXT thread_context = {CONTEXT_DEBUG_REGISTERS};
            thread_context.Dr0 = func_addr;
            thread_context.Dr7 = (1 << 0);
            SetThreadContext(hMainThread, &thread_context);
            CloseHandle(hMainThread);
        }
        CloseHandle(hTool32);
    }
}

int APIENTRY DllMain(HMODULE hModule, DWORD reason, LPVOID reserved) {
    if(reason == DLL_PROCESS_ATTACH) {
        DisableThreadLibraryCalls(hModule);
        if(AllocConsole()) {
            freopen("CONOUT$", "w", stdout);
            SetConsoleTitle(L"Console");
            SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
            printf("DLL loaded.\n");
        }
        set_breakpoints();
    }
    return TRUE;
}
```

The DLL entry point sets up a console for debugging purposes and sets the breakpoints. The `set_breakpoints` function works by taking a snapshot of all threads on the system and iterating through them until the threads of the target process are found. The thread with the earliest creation time within the application is the main thread. The handle for this thread is kept so the debug registers can be set up. Once the main thread is found, the actual SEH handler can be installed. `SetUnhandledExceptionFilter` sets the `ExceptionFilter` function as the top-level exception handler so it will be called prior to all others (but not prior to VEHs). The CONTEXT structure is set up with ContextFlags being CONTEXT_DEBUG_REGISTERS, DR0 is set to the desired address, and DR7 is set to a global enable level for the address in DR0. The CONTEXT of the main thread is then set to this new context and the breakpoints are now active. When an exception is raised, `ExceptionFilter` checks to see whether the exception occurred at the desired address. If so, the exception is handled and now the context record (containing, among other things, the values of all registers and flags when the breakpoint was hit). Since the function sets up a standard BP-based frame, the parameters can all be retrieved through ESP (since the stack frame was not set up yet when the breakpoint was hit). All registers and parameters can then be inspected and/or modified as shown in `print_parameters` and `modify_text`. The pictures below show how this looks at run-time:



An important thing to note in the code is the need of the stub function. This stub function contains the first instruction of the function that has the breakpoint on it. Then it jumps one byte past the breakpoint address, where the next instruction starts. This is needed because if EIP is not modified, the exception will be raised again once the handler finishes and an infinite loop will occur. Making a stub function is a quick workaround to that problem. That is pretty much all to it in terms of SEH. Removing the breakpoints is as simple as clearing the debug registers in the main thread (not shown in the code for simplicity).
The technique does not differ much for VEH. To use VEH instead of SEH, only the following modifications need to be made:

Change `(void)SetUnhandledExceptionFilter(ExceptionFilter);` to `(void)AddVectoredExceptionHandler(1, ExceptionFilter);`. The VEH can be removed by clearing the debug registers and calling `RemoveVectoredExceptionHandler`. An important thing to note is that `AddVectoredExceptionHandler` returns a handle to the exception handler that I chose to ignore for the sake of showing the technique and saving space. However, this return value is needed if the handler is to be removed at a later time since `RemoveVectoredExceptionHandler` requires it.
Code and sample application for SEH/VEH can be found here
A downloadable PDF of this post can be found here