Using Hardware Breakpoints in a Loader for Packed Executables to Circumvent Protective Mechanisms

Grant “afman” Curell
1. Overview

This paper is meant for pedagogical purposes only. It is meant primarily as in introduction to loaders and their many uses. It discusses many topics to include dll injection, vectored exception handling, and as the title implies, using hardware breakpoints in the WINAPI. Hopefully, it proves a useful guide to others just entering the field of reverse engineering.

2. Introduction to the Problem

I. What is a packed executable?

A packed executable is an executable file that has been compressed in some capacity. The original intent of packing executables was to reduce the amount of space they consumed. As time progressed, packed executables started to see use as a mechanism to defeat reverse engineers. An executable is traditionally packed by some algorithm and a stub is appended to the packed section of the executable that unpacks the executable at run time. This is also known as a self-extracting executable. Several commercial products exist to automatically pack an executable and append the corresponding unpacking stub. Several unpackers exist, which simply reverse the process. An unpacker typically utilizes a signature database to identify the method by which an executable was packed and unpack the target executable; returning it to its original state.

II. Why does this make reverse engineering more difficult?

In the modern day saving a couple of kilobytes of space is not the concern it was in the 90’s. In fact outside the area of mobile computing or embedded systems a couple of kilobytes compared to the terabytes of available storage is an altogether ignorable amount. So why then are executables still packed? Now, executable packing
exists primarily to thwart attempts to reverse engineer the executable by obscuring its contents. When a packer compresses an executable it essentially obscures the inner workings of that executable making it completely unreadable by a disassembler. In fact when Immunity Debugger’s disassembler attempts to disassemble a packed executable it prompts the user with an error message:

![Entry Point Alert](image)

This message occurs because the uncompressed, unpacking stub that was appended by the packer, is not inside the code section of the executable file as defined by the PE/COFF header [1]. Since the goal of this particular reverse engineering endeavor is to modify the program in some way that causes the program to always run in a registered or authorized state this presents a significant problem in that the code that needs to be modified is hidden before runtime. Several approaches exist to defeat a packing protection mechanism.

### III. How to beat executable packing?

There exist a couple of well-known ways to beat packed executables. The simplest of methods is to unpack the executable with one of the many freeware unpackers. Once the executable is in its original state a reverse engineer is free to modify it in any way he/she chooses. However, this presents a problem. What if there is no unpacker for a specific compressed executable? Another method is for a reverse engineer to study and understand the uncompressed, unpacking stub attached to the executable and modify it to manipulate the code as it is unpacked. This also presents a problem in that the stub might be highly complex and difficult to
understand. That aside, there also exists the matter of modifying the stub perfectly to produce the correct code. Finally, what if there is a CRC check performed at some point on the executable? If the stub was modified the CRC check is likely to fail. There is another possibility. Why not let the stub completely unpack the executable and then modify it at runtime? This defeats many rudimentary CRC check because the executable isn’t modified in any way before runtime. It also saves the reverse engineer the trouble of attempting to understand what could be a very complex unpacking routine. All the loader has to do is open the target program is a child process, wait for the appropriate time, and modify the target program in memory. This brings around the crux of this article. How does the loader know when to patch?

IV. Deciding when to patch

This is the pivotal issue in the loader scheme. In this paper a pedagogical example is used called Duelist Crackme #5 [2]. Duelist Crackme #5, from here on out referred to as due5, is packed by some arbitrary method that for the purposes of this paper are not important. The goal in this paper is to remove the nag screen and cause the program to run in registered mode. Below is a picture of the nag screen and the running program:
One common method of deciding when to use the loader to patch the executable is to wait for user input to be idle [3]. This is one of the most simplistic methods. All that is required is for the loader to make use of the WaitForInputIdle function provided by the WINAPI. Another method is to wait for a window associated with the target program to appear [4]. However, these methods both present a problem in the case of due5. They both require a window to already exist. In the case of due5, the goal is to remove the nag screen on startup, which will appear before patching in the case of the previous two methods. Novice users might recommend using a software breakpoint to stop execution at just the right point. However, what they don’t realize is that software breakpoints, as the name implies, are implemented by modifying the program (software) by adding an instruction that causes an interrupt. However, a software breakpoint will not work in this scenario because the breakpoint instruction will, in all likelihood, be modified when the executable is uncompressed. This is where hardware breakpoints come in to play.

3. Problem Solution

I. Using hardware breakpoints in the WINAPI

Officially, the WINAPI only supports read/write hardware breakpoints. However, it is possible to implement hardware breakpoints manually in the WINAPI via the GetContext and SetContext instructions [5]. In this scenario, a hardware breakpoint could be placed on the correct instruction, that when executed will stop execution and wait for the loader to modify the program. Intel provides four registers to hold the addresses of hardware breakpoints and two registers to hold their settings [6]. These breakpoints do not modify the code in any way. A hardware breakpoint is triggered by a read/write from a specified memory location, at which point a debug exception is generated. Machine specific registers handle monitoring the debug registers. Below is the code to set a hardware breakpoint on execution:
HANDLE SetHardwareBreakpoint(HANDLE hThread, HWBRK_TYPE Type, HWBRK_SIZE Size, void* s) 
{
    //Set constants
    HWBRK* h = new HWBRK; //Create a new hardware breakpoint
    h->a = s; //Address of the breakpoint
    h->Size = Size; //Size of the breakpoint to be set
    h->Type = Type; //Type of breakpoint to set (read/write/execute)
    h->hT = hThread; //The thread to set the breakpoint in
    h->hEv = CreateEvent(0,0,0,0);
    h->Opr = 0; // 0 specifies set breakpoint 1 specifies remove breakpoint

    //Create a remote thread to set the hardware breakpoints. Hardware
    //breakpoints are not global in windows. They are specific to the context of one thread.
    HANDLE hY = CreateThread(0,0,th,(LPVOID)h,0,0);

    //Wait for the breakpoint instance's event object to be placed in a signaled state.
    WaitForSingleObject(h->hEv,INFINITE);
    CloseHandle(h->hEv);
    h->hEv = 0;

    if (hThread == GetCurrentThread())
    {
        CloseHandle(h->hT);
    }
    h->hT = hThread;

    if (!h->SUCC)
    {
        delete h;
        return 0;
    }

    return (HANDLE)h;
}

This code is directly from [5]. The code creates a HWBRK instance that contains all the information required to create a hardware breakpoint. It then creates a thread that actually sets the hardware breakpoint and performs cleanup. Below is the code to the th function, called by CreateThread above:

static DWORD WINAPI th(LPVOID lpParameter)
{
    //lpParameter is passed via the CreateThread parameter so it has
    //to be typecasted back to a HWBRK type
    HWBRK* h = (HWBRK*)lpParameter;
    int j = 0;
    int y = 0;

    //Suspend the target thread so the breakpoint can be set. As a
    //reminder SuspendThread returns a threads SuspendCount
    j = SuspendThread(h->hT);
    y = GetLastError();

    CONTEXT ct = {0};

//Get the context of the DRX register set.
c.t.ContextFlags = CONTEXT_DEBUG_REGISTERS;

//If GetThreadContext succeeds it returns a non-zero number
//otherwise it returns 0
j = GetThreadContext(h->hT, &ct);
y = GetLastError();

int FlagBit = 0;

//Set all debug registers as unused
bool Dr0Busy = false;
bool Dr1Busy = false;
bool Dr2Busy = false;
bool Dr3Busy = false;

//Each of these correspond to the Global
//breakpoint enable registers.
if (ct.Dr7 & 1)
  Dr0Busy = true;
if (ct.Dr7 & 4)
  Dr1Busy = true;
if (ct.Dr7 & 16)
  Dr2Busy = true;
if (ct.Dr7 & 64)
  Dr3Busy = true;

//Opr set to 1 means remove breakpoints
if (h->Opr == 1)
{
  // Clear debug registers
  if (h->iReg == 0)
  {
    FlagBit = 0;
    ct.Dr0 = 0;
    Dr0Busy = false;
  }
  if (h->iReg == 1)
  {
    FlagBit = 2;
    ct.Dr1 = 0;
    Dr1Busy = false;
  }
  if (h->iReg == 2)
  {
    FlagBit = 4;
    ct.Dr2 = 0;
    Dr2Busy = false;
  }
  if (h->iReg == 3)
  {
    FlagBit = 6;
    ct.Dr3 = 0;
    Dr3Busy = false;
  }
  ct.Dr7 &= ~(1 << FlagBit);
}
//If opr was 0 then the below sets the first available
//debug register and marks it as used
else
{
if (!Dr0Busy)
{
    h->iReg = 0;
    ct.Dr0 = (DWORD_PTR)h->a;
    Dr0Busy = true;
}
else
if (!Dr1Busy)
{
    h->iReg = 1;
    ct.Dr1 = (DWORD_PTR)h->a;
    Dr1Busy = true;
}
else
if (!Dr2Busy)
{
    h->iReg = 2;
    ct.Dr2 = (DWORD_PTR)h->a;
    Dr2Busy = true;
}
else
if (!Dr3Busy)
{
    h->iReg = 3;
    ct.Dr3 = (DWORD_PTR)h->a;
    Dr3Busy = true;
}
else
{
    h->SUCC = false;
    j = ResumeThread(h->hT);
    y = GetLastErrorCode();
    SetEvent(h->hEv);
    return 0;
}
ct.Dr6 = 0;
int st = 0;
if (h->Type == HWBRK_TYPE_EXECUTE)
    st = 0;
if (h->Type == HWBRK_TYPE_READWRITE)
    st = 3;
if (h->Type == HWBRK_TYPE_WRITE)
    st = 1;
int le = 0;
if (h->Size == HWBRK_SIZE_1)
    le = 0;
if (h->Size == HWBRK_SIZE_2)
    le = 1;
if (h->Size == HWBRK_SIZE_4)
    le = 3;
if (h->Size == HWBRK_SIZE_8)
    le = 2;

SetBits(ct.Dr7, 16 + h->iReg*4, 2, st);
SetBits(ct.Dr7, 18 + h->iReg*4, 2, le);
SetBits(ct.Dr7, h->iReg*2, 1, 1);
}
ct.ContextFlags = CONTEXT_DEBUG_REGISTERS;
j = SetThreadContext(h->hT, &ct);
This code is slightly more complex than the code from SetHardwareBreakpoint. The HWBRK instance that contains the data for the hardware breakpoint is passed in through lpParameter. The thread containing the hardware breakpoint is suspended while the code modifies the thread's context. It grabs the thread’s context and first checks to see if any of the debug registers are currently in use. [6] details the use of Dr7 and the operation of the debug registers. It then checks to see if th was called to remove a hardware breakpoint or to set a hardware breakpoint. It then proceeds to check which debug registers are unused and sets or removes the hardware breakpoint accordingly. Finally it sets the thread’s context and resumes the thread’s normal execution.

III. Finding where to place the hardware breakpoint

Finding where to place the hardware breakpoint only requires a debugger. In this paper Immunity Debugger was used. All that is required is a location in the original code. For due5 it is only a matter of executing the program under a debugger. By examining the original PE header the original entry point into the code section can be found. The below picture is a screenshot of a dump of the PE header of due5 from Immunity Debugger:
From this it is possible to deduce that the original code should appear at address 0x00401000. This can be confirmed by looking at the memory address 0x00401000:

Clearly, this is not readable code. This occurs because Immunity realized that the executable was gibberish and stopped analyzing the executable for code. By right clicking on the first instruction, clicking analysis, and then analyze as code Immunity will reanalyze the executable and produce this:

It does not take long to notice a little bit below the section that produces the nag message on startup:
By changing the above to NOPs the nag box is effectively destroyed. That only leaves the word unregistered in the main window. The code for the main window is here:

By changing the lParam parameter from due-cm5.0040205C to due-cm5.00402050 the word Unregistered is changed to Registered:

For the purposes of this paper the hardware breakpoint was placed on 0x004010C1. The hardware breakpoint was set to break on execute. This raises a question though; how does the exception generated by the hardware breakpoint get caught?

IV. Using vectored exception handling to catch the hardware breakpoint

The main issue with using a hardware breakpoint is that the loader needs some way to catch the exception generated by the hardware breakpoint. This is accomplished by using vectored exception handling [7]. This raises yet another issue. If the application registers a vectored exception handler before the vectored exception loader the exception is never caught. Unfortunately, this is unavoidable. Therefore, it is best to place the hardware breakpoint as close to the entry point of the program as possible. In this pedagogical example this is not a concern since the application does not make use of vectored exception handling. The
other issue this raises is how will the loader register the vectored exception handler? In this case there needs to be code running within the target process’ address space. In this instance the loader uses a dll running in the process’ address space to register the handler. The dll is loaded via dll injection [8]. This is discussed in further detail in the next section. Below is the code for the dll that will register the vectored exception handler:

```c
void SetHandler() {
    //Add the vectored exception handler to the front of the VEH linked list.
    AddVectoredExceptionHandler(1, BreakpointHandler);
}
```

... 

```c
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL, // handle to DLL module
    DWORD fdwReason,   // reason for calling function
    LPVOID lpReserved ) // reserved
{
    //If the DLL has just loaded...
    if(fdwReason == DLL_PROCESS_ATTACH) {
        SetHandler();
    } else if (fdwReason == DLL_PROCESS_DETACH) {
        //Remove the vectored exception handler
        RemoveVectoredExceptionHandler(BreakpointHandler);
    }
    return TRUE; // Successful DLL_PROCESS_ATTACH.
}
```

All this code does is register the handler as soon as the dll is loaded into the process’ address space. As soon as the hardware breakpoint is reached the vectored exception handler BreakpointHandler will be called.
V. Patching the process

Patching the process only requires WriteMemory to write over the original instructions with whatever the user specifies. Below is the code inside of the dll that is called once the breakpoint is reached that patches the process:

```c
LONG __stdcall BreakpointHandler(struct _EXCEPTION_POINTERS *pExceptionInfo) {

    //Causes the compiler not to throw a warning. The compiler ends up optimizing this //out since it doesn't really do anything.
    UNREFERENCED_PARAMETER(pExceptionInfo);

    //Get the exception code from the EXCEPTION_POINTERS structure.
    LONG exceptionCode = pExceptionInfo->ExceptionRecord->ExceptionCode;

    //Check if our breakpoint was hit.
    if (exceptionCode == STATUS_SINGLE_STEP) {
        //Patch the program
        PatchProgram(GetCurrentProcess(), GetCurrentThread());

        //Get the event object held by the loader process
        hwbrkEvent = OpenEvent(EVENT_MODIFY_STATE, FALSE, (LPCWSTR)L"hwbrkEvent");

        //Make sure the event object was successfully opened.
        if (hwbrkEvent == NULL) {
            MessageBox(NULL, (LPCWSTR)L"Event failed to open.", (LPCWSTR)L"It's working...", MB_OK | MB_ICONEXCLAMATION);
            exit(0);
        }

        //Set the event to a signaled state.
        SetEvent(hwbrkEvent);

        hwbrkEvent2 = CreateEventA(
            NULL,
            TRUE,
            FALSE,
            "hwbrkEvent2");

        if (hwbrkEvent2 == NULL) {
            MessageBox(NULL, (LPCWSTR)L"Failed to create event.", (LPCWSTR)L"It's working...", MB_OK | MB_ICONEXCLAMATION);
            exit(0);
        }

        WaitForSingleObject(hwbrkEvent2, INFINITE);

        //CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)removeHandler, NULL, NULL, NULL);
    }
}
```
CloseHandle(threadHandle);

    //Resume execution of the program.
    return EXCEPTION_CONTINUE_EXECUTION;
}

//Since this wasn't our breakpoint call the next VEH or go to the SEH chain.
return EXCEPTION_CONTINUE_SEARCH;
}

BOOL PatchProgram(HANDLE hProcess, HANDLE hThread) {

    //This is the number of patches that will occur to the program.
    const int patchNum = 23;

    //These are the patches to be applied to the program.
    Patch crk[patchNum];
    crk[0] = Patch(0x004010C1, 0x90);
    crk[1] = Patch(0x004010C2, 0x90);
    crk[2] = Patch(0x004010C3, 0x90);
    crk[3] = Patch(0x004010C4, 0x90);
    crk[4] = Patch(0x004010C5, 0x90);
    crk[5] = Patch(0x004010C6, 0x90);
    crk[6] = Patch(0x004010C7, 0x90);
    crk[7] = Patch(0x004010C8, 0x90);
    crk[8] = Patch(0x004010C9, 0x90);
    crk[9] = Patch(0x004010CA, 0x90);
    crk[10] = Patch(0x004010CB, 0x90);
    crk[11] = Patch(0x004010CC, 0x90);
    crk[12] = Patch(0x004010CD, 0x90);
    crk[13] = Patch(0x004010CE, 0x90);
    crk[14] = Patch(0x004010CF, 0x90);
    crk[15] = Patch(0x004010D0, 0x90);
    crk[16] = Patch(0x004010D1, 0x90);
    crk[17] = Patch(0x004010D2, 0x90);
    crk[18] = Patch(0x004010D3, 0x90);
    crk[19] = Patch(0x004010D4, 0x90);
    crk[20] = Patch(0x004010D5, 0x90);
    crk[21] = Patch(0x004010D6, 0x90);
    crk[22] = Patch(0x00401131, 0x50);

    //Find the specified memory locations and patch them.
    unsigned long byteswritten[patchNum];
    unsigned long bytesread[patchNum];

    //Zero out the arrays
    for (int i=0; i<patchNum; i++) {
        bytesRead[i] = 0;
        byteswritten[i] = 0;
    }

    for (int idx=0; idx<patchNum; idx++) {
        //Read in the data to be patched
        ReadProcessMemory(hProcess,
                          (LPVOID)(crk[idx].address),
                          (LPVOID)(&crk[idx].orig)),
                          1,
                          &bytesread[idx]);
        //Handle any errors that occured
        if(bytesread[idx] == 0)
The code first checks to see if the exception caught by the vectored exception handler is of type STATUS_SINGLE_STEP. If it is, PatchProgram is immediately called which uses WriteProcessMemory to write the specified data into the active process’ memory. After patching the process’ memory, the dll alerts the loader via a global event that execution is ready to resume. The loader then calls RemoveHardwareBreakpoint. The code for RemoveHardwareBreakpoint is below:
Counter to its name RemoveHardwareBreakpoint does not actually remove the hardware breakpoint. It simply sets EIP to the middle of the NOP slide created when PatchProcess overwrites the data for the dialog window with NOPs. Admittedly, the original code did remove the hardware breakpoint. However, in testing even with the debug registers zeroed out the breakpoint continually triggered the exception handler. After several days of searching for possible solutions the decision was made to cut the corner and just change EIP to move on to the next instruction rather than remove the hardware breakpoint. After the process is patched and EIP changed, the execution of the target process continues. The only thing left to do is implement the loader and tie together all the code already discussed.
VI. Implementing the loader

The loader’s job is to use CreateProcess to create the target process, set/remove the hardware breakpoint and inject the dll with the patches into the target process. Below is the code for the loader:

```c
#include <Windows.h>
#include <stdio.h>
#include "HwBrk.h"

void InjectDLL(HANDLE Proc);
HANDLE hwbrkEvent, hwbrkEvent2;

BOOL main(int argc, char* argv[]) {
    PROCESS_INFORMATION process_info;
    STARTUPINFO startup_info;

    //0 out the area for our process' structures
    ZeroMemory( &startup_info, sizeof(startup_info) );
    startup_info.cb = sizeof(startup_info);
    ZeroMemory( &process_info, sizeof(process_info) );

    //Grab the process to be loaded. (In this case that should be
    //the duelist 5 executable file due-cm5.exe
    if(!::CreateProcess(  
        "C:\Users\afman\Documents\Visual Studio 2010\Projects\Loader\Debug\due5.exe",  //LPCTSTR lpApplicationName
        NULL,  //LPCTSTR lpCommandLine
        NULL,  //LPSECURITY_ATTRIBUTES lpProcessAttributes
            //Process handle not inheritable
        NULL,  //LPSECURITY_ATTRIBUTES lpThreadAttributes
            //Thread handle not inheritable
        FALSE,  //BOOL bInheritHandles
            //Set handle inheritance to FALSE
        CREATE_SUSPENDED,  //DWORD dwCreationFlags
            //Use suspended creation flags
        NULL,  //LPVOID lpEnvironment
            //Use parent's environment block
        NULL,  //LPCTSTR lpCurrentDirectory
            //Use parent's starting directory
        &startup_info,  //LPSTARTUPINFO lpStartupInfo
            //Pointer to STARTUPINFO structure
        &process_info)  //LPPROCESS_INFORMATION lpProcessInformation
            //Pointer to PROCESS_INFORMATION struture.
    ) {
        //If the process can’t be created return an error message.
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return FALSE;
    }
```
void* ptr = (void*)0x004010C1;

printf("Setting hardware breakpoint...\\n");

HANDLE hardwareBreakpoint = SetHardwareBreakpoint(process_info.hThread, HWBRK_TYPE_EXECUTE, HWBRK_SIZE_1, ptr);

//Create an event that will be signaled when we are ready to remove the hardware breakpoint.
hwbrkEvent = CreateEvent(
    NULL,
    TRUE,
    FALSE,
    "hwbrkEvent");

if (hwbrkEvent == NULL)
{
    printf("CreateEvent error: %d\\n", GetLastError());
    return FALSE;
}

printf("Starting injection process...\\n");

//Inject the DLL into the remote thread.
InjectDLL(process_info.hProcess);

printf("Resuming thread...\\n");

//Resume the thread
ResumeThread(process_info.hThread);

printf("Waiting to remove breakpoint...\\n");

//Wait for the event object to be signaled by the injected dll.
WaitForSingleObject(hwbrkEvent, INFINITE);

//Remove the hardware breakpoint
RemoveHardwareBreakpoint(hardwareBreakpoint);

//Tell the injected dll that the hardware breakpoint has been removed
hwbrkEvent2 = OpenEvent(EVENT_MODIFY_STATE, FALSE, "hwbrkEvent2");

if (hwbrkEvent2 == NULL)
{
    printf("Event failed to open.");
    return FALSE;
}

SetEvent(hwbrkEvent2);

//Set the event back to unsigned
//ResetEvent(hwbrkEvent);

printf("Closing handle to process...\\n");

//Close the handle and return.
CloseHandle(process_info.hProcess);

return TRUE;
void InjectDLL(HANDLE Proc)
{
    //Declare constants.
    char buf[50]="\0";
    LPVOID RemoteString, LoadLibAddy, SetHandler;
    LPCSTR DLL_NAME = "HwBrkDLL.dll";

    printf("Getting LoadLibrary handle...\n");

    //Get the address of the LoadLibrary function in kernel32.dll
    LoadLibAddy = (LPVOID)GetProcAddress(GetModuleHandle("kernel32.dll"), "LoadLibraryA");

    printf("Creating remote string with the name of the DLL to be injected...\n");

    //Allocate space in the remote process to store the name of the library to be loaded.
    RemoteString = (LPVOID)VirtualAllocEx(Proc, NULL, strlen(DLL_NAME) + 1, MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);

    printf("Writing string to remote memory...\n");

    //Write the name of the DLL into the allocated memory.
    WriteProcessMemory(Proc, (LPVOID)RemoteString, DLL_NAME, strlen(DLL_NAME) + 1, NULL);

    printf("Creating remote thread to load DLL...\n");

    //Create a remote thread in the victim process that loads the library.
    CreateRemoteThread(Proc, NULL, NULL, (LPTHREAD_START_ROUTINE)LoadLibAddy, (LPVOID)RemoteString, NULL, NULL), INFINITE);

    //Free the memory we used for the string.
    VirtualFreeEx(
        Proc,
        RemoteString,
        strlen(DLL_NAME) + 1,
        MEM_RELEASE);
}

The first thing the loader does is create the target process. In this case the target process is due5. After creating the process it sets the hardware breakpoint inside the context of the only thread in the target process. After creating the hardware breakpoint InjectDLL is called. InjectDLL uses the CreateRemoteThread method of dll injection [9]. This uses CreateRemoteThread in the target process to call the function LoadLibrary in kernel32.dll to load the dll into the target process’ memory. The only problem with this method is that there isn’t a way to pass the name of the dll to be loaded to LoadLibrary. The workaround for this problem is to use VirtualAllocEx to allocate memory in the target process and write the name of the dll to that
memory. When CreateRemoteThread is called the address of the string is used as an argument passed to LoadLibrary.

After the dll is injected into the memory of the target process the loader waits until the dll sets the event that causes RemoveHardwareBreakpoint to be called. Once that is complete the loader cleans up and exits and the target process runs completely free of any interference. The only modification that remains is the dll will still reside in the memory of the target process. The resulting product should result in a registered window being the only window displayed by the program:

![Image of registered window](image)

4. Closing Thoughts

While this is certainly not the most efficient way to implement a loader it introduces several useful concepts. Readers interested in this topic might find it interesting to try implementing the same loader as a kernel mode driver. This is a much more “correct” way of implementing the loader. It avoids the problem of having programs register a vectored exception handler in front of the one created by the loader. Hopefully, this helped others also experimenting with the WINAPI or learning to using loaders.
References


