

- [0174] 5. The callback function `GetMsgProc`, supplied by `HELPER1.DLL`, waits for a `WM_NULL` (or equivalent) message. This function always returns with a call to `::CallNextHookEx`.
- [0175] 6. When `GetMsgProc` receives the anticipated message, it calls `SubclassInnocentApp` which simply calls `::SetWindowLong (. . . , GWL_WNDPROC, . . . )` on either the victim's window or one of its children, passing the address of `NewVictimProc` while storing the returned original procedure address for a later use.
- [0176] 7. The callback function `NewVictimProc` does whatever it wishes upon receiving the messages it wishes to divert. Other messages are passed to the original procedure with `::CallWindowProc`.
- [0177] 8. Clean-up procedures are not covered here.
- [0178] Offender, Mechanism #2
- [0179] 1. The offender launching program, `LAUNCHER2.EXE`, initializes its connection to the windows-hooks stack and does other common startup things.
- [0180] 2. `LAUNCHER2.EXE` calls `CatchInnocentApp`, a function that is supplied by `HELPER2.DLL`, passing it its own current thread ID.
- [0181] 3. Function `CatchInnocentApp` calls `::SetWindowsHookEx(WH_KEYBOARD, . . . )` on all threads on this 'desktop' object (last argument is 0). On advanced versions of windows, calling `::SetWindowsHookEx(WH_KEYBOARD_LL, . . . )` can provide low-level keyboard input events.
- [0182] 4. The callback function `KeyBoardProc` supplied by `HELPER2.DLL`, waits for a keyboard message. It also checks to see that the current thread ID is not the thread ID of `LAUNCHER2.EXE`. This function always returns with a call to `CallNextHookEx`.
- [0183] 5. When `KeyBoardProc` receives a keyboard message, it can do with it whatever it wishes. This would typically include processing the keyboard status and the thread current language setting to interpret the exact meaning of the key(s) pressed, then sending the information out to an unauthorized person.
- [0184] 6. Clean-up procedures are not covered here.
- [0185] Defender for Offender Mechanism #2
- [0186] 1. The launching program, `DEFENDER.EXE`, initializes its connection with the windows-hooks stack and does other common startup things.
- [0187] 2. `DEFENDER.EXE` calls to `CatchBadApp`, a function which is supplied by `ASSITANT.DLL`, passing it its own current thread ID.
- [0188] 3. Function `CatchBadApp` calls `::SetWindowsHookEx(WH_DEBUG, . . . )` on all threads on this 'desktop' object (last argument is 0).
- [0189] 4. It then calls `::SetWindowsHookEx(WH_GETMESSAGE, . . . )` on all threads on this 'desktop' object (last argument is 0). There are now two hooks managed by `ASSISTANT.DLL` (the purpose of the second hook will be apparent thereafter).
- [0190] 5. The callback function `DebugProc`, supplied by `ASSISTANT.DLL`, waits for a keyboard hook notification, `WH_KEYBOARD`. This function always returns with a call to `::CallNextHookEx`.
- [0191] 6. When `DebugProc` receives an anticipated notification (in this case, a keyboard), the OS also supplies it with a `::DEBUGHOOKINFO` structure, so it can retrieve both the thread ID of the thread containing the filter function and the thread ID of the thread that installed the debugging hook. (Important note: this step was demonstrated on Win9x but not on NT. See the notes in a later section for more details).
- [0192] 7. Now `DebugProc` calls `::PostThreadMessage` on the installer thread ID, passing it a user-defined message, `WM_DEFENDER`. It also supplies the containing thread ID as `LPARAM` as a hint for the receiver.
- [0193] 8. `GetMsgProc` waits for a `WH_MSG` notification of message type `WM_DEFENDER`. When received, it calls `::GetModuleFileName` to retrieve the bad application's name (and full path).
- [0194] 9. Now `GetMsgProc` can do whatever it wishes with the offending program, acting from within the thread of the offending program. The simplest act would be posing a message to the user, asking him if he wishes to close the program and letting him know the name and path of the suspected offender. If the user decides to close the suspect, `GetMsgProc` would simply call `::ExitThread` for a graceful exit. Of course there are many other, more sophisticated acts that may be taken.
- [0195] 10. Clean-up procedures are not covered here.
- [0196] Some Notes
- [0197] 1. A part of the mechanism that is described herein is covered in well known programming books and in other publicly available articles. These, however, are mainly concerned with the task of bringing a DLL into the address space of another process (or 'injecting' it)—not with the malicious acts that may follow, nor in the ways of detecting such acts—the later being the main concern of the present invention.
- [0198] 2. On Windows NT and its descendants (like Windows 2000), the system seems not to provide the offender thread ID with the `DEBUGHOOKINFO` structure. This behavior seems to be inconsistent with the current official on-line documentation that also seems to state that the `DEBUGHOOKINFO` structure is not implemented on Win9x, a statement that is apparently imprecise. Neglecting to handle these (apparently misdocumented) details will lead to a lame implementation of the defender under Windows NT and its descendants, while the previously described offender goes undisturbed.
- [0199] 3. The concepts are not dependent on the previously described specific OS-supplied API for detecting the presence of a new DLL or the invocation of some procedures; using such a mechanism is just a convenience that keeps this example simple. Many complementary tools and mechanisms exist, and more may be devised for fulfilling this task.
- [0200] 4. The importance of retrieving and storing different thread IDs is due to the fact that the mechanism