SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **40**:567–584
Published online 20 April 2010 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.973

Extending applications using an advanced approach to DLL injection and API hooking

J. Berdajs*,† and Z. Bosnić

Faculty of Computer and Information Science, University of Ljubljana, Tržaška cesta 25, 1000 Ljubljana, Slovenia

SUMMARY

When programmers need to modify third-party applications, they frequently do not have access to their source code. In such cases, DLL injection and API hooking are techniques that can be used to modify applications without intervening into their source code. The commonly used varieties of injection and hooking approaches have many practical limitations: they are inconvenient for a programmer to implement, do not work reliably in conjunction with all applications and with certain low-level machine instructions. In this paper we present two novel approaches to DLL injection and API hooking, which we call *Debugger-aided DLL injection* and *Single Instruction Hooking*. Our approaches overcome the limitations of the state-of-the art approaches. Despite incurring greater execution times, our approach allows extending of the applications in situations where the comparable approaches fail. As such, it has a notable practical value for beneficial practical applications of injection and hooking approaches, which are present in malware detection programs and computer security tools. Copyright 2010 John Wiley & Sons, Ltd.

Received 20 May 2009; Revised 10 October 2009; Accepted 3 March 2010

KEY WORDS: application extensions; programming; DLL injection; API hooking; detours

1. INTRODUCTION

To integrate new services into third-party applications, developers often face the need to do so without having the original application's source code at their disposal. To allow extensions, applications sometimes provide the means of interfacing with them, such as plug-in and similar interfaces. Although the use of such interfaces is the preferred way for performing extensions, they offer only as much programming freedom as they were designed to provide, which is often not enough to satisfy a creative developer's needs.

Since the source code of the majority of applications is not publicly available, especially for applications designed for commercial operating systems (e.g. the Windows platform), a programmer may not have the possibility to extend an application by modifying its source code. There are, however, a few alternatives. First, a programmer can use lower-level programming approaches to achieve this goal, such as modifying the application's machine code in case when the required modifications are not too extensive. In practice, this approach is used to fix small bugs, perform trivial modifications and even bypass security measures in the pirate community. However, when bigger application modifications are needed, the use of machine code-level programming becomes increasingly difficult or even impossible due to low code manageability. Focusing on the Windows

*Correspondence to: J. Berdajs, Faculty of Computer and Information Science, University of Ljubljana, Tržaška cesta 25, 1000 Ljubljana, Slovenia.
†E-mail: mrbrdo@gmail.com

Contract/grant sponsor: Publishing Arts Research Council; contract/grant number: 98-1846389

Copyright 2010 John Wiley & Sons, Ltd.

IPR2023-00895
KASPERSKY EX1030 Page 1

operating system, as an alternative, a combination of two approaches can be used to perform the task at question more efficiently: They are Dynamic Link Library (*DLL*) *injection* and *function hooking*. DLL injection [1] is a concept of loading code into the address-space of the target application through a DLL, making subsequent interactions with the application's memory and functions easier. Function hooking [2], also referred to as code or Application Programming Interface[‡] (API) hooking, is an approach used to modify an application's behavior by making it use as an arbitrary function instead of the originally intended one (e.g. the result of the function hooking could cause that the click of a button e-mails the displayed text instead of storing it into a file).

In this paper, we present a novel approach to extending the existing applications without given source code that involves a combination of DLL injection and function hooking. In contrast to the existing similar methods, our approach overcomes limitations that prevent other approaches to be used under certain conditions, e.g. injection in the initialization stage of an application (the current DLL injection techniques fail to inject a DLL during the initialization stage of an application, before the application starts its normal execution). Additionally, when a function contains specific code, it can be difficult or even impossible to hook it using the conventional approaches. In our work, we address these issues and discuss on how to bypass these limitations.

The paper is organized as follows. An overview of the related work is given in Section 2. This section comprises of descriptions of alternative DLL injection and API hooking approaches, including the description of *Detours* hooking, which is the basis of our approach. We present our method in Section 3 where we describe our novel implementation of DLL injection and API hooking and outline the differences to the elementary hooking approach using *Detours*. In Section 4, we evaluate our approach by comparing it with the alternative approaches. We conclude the paper in Section 5 with the ideas for potential future work.

2. RELATED WORK

Successful utilization of *DLL injection* and *API hooking* approaches strongly relies on understanding of the underlying operating system's internals and API functions. Since the development of our approach is operating system-dependent, in this paper we focus on the Windows NT operating system architecture. Some of the pioneers in the area of documenting interesting aspects of Windows API functions used in some API hooking techniques were Alpaydin, Maxey and Pietrek [4, 5] and Nebbett [6], consequently enabling faster development of initial API hooking approaches by documenting the Windows Portable Executable (PE) format (the format in which executable files are structured on disk) [7–9], as well. The following subsections present the work in the related areas.

2.1. DLL injection

One of the first publications related to Windows NT DLL injection [1] describes an approach for loading a DLL into an arbitrary running application. This is performed by remotely executing code in the application via an API function implemented in the Windows NT family. The approach allocates memory for necessary data structures and a thread function in the target process. By starting a remote thread using *CreateRemoteThread*, the DLL is loaded into the process using an API function. Throughout the years, this basic approach has remained almost unchanged and is still the preferred way for performing DLL injection. In the remainder of the paper, we will refer to this approach as the *CreateRemoteThread approach*. Some practical discussions [10] have advocated DLL injection to be used for various purposes, such as extending the Windows task manager [11], spying on other applications (WinSpy) [12], performing and preventing code injection attacks [13] and application isolation [14], which is employed by some anti-virus software.

[‡]An API is a set of routines, data structures, object classes and/or protocols provided by libraries and/or operating system services in order to support the building of applications [3].

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 2

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Alternative approaches to injection include:

(i) *Creating a Proxy DLL*: When an application requires a DLL, Windows tries to locate it in the directories by the order of their search precedence. If we were to put a *proxy DLL* (a DLL that allows calls to be intercepted, data altered, and even new objects created) with the same name as the original DLL in a directory with higher precedence, the proxy DLL would be loaded instead of the original one. The DLL can then provide alternative implementations of the original DLL functions. Although this approach is simple to implement and requires little programming knowledge, it can be tedious to implement in some cases since it requires redirection of all the functions that the original DLL implements.

Changes in the original DLL (e.g. by updates) can also present a problem for proxy DLLs (depending on the implementation). It should be noted that more recent versions of Windows (e.g. 2000 SP2, XP SP1, Vista) introduced a change in the precedence of DLL search directories, which makes the priority of the directory of the executed application lower than the priority of system directories (unlike previous versions, in which the directory of the executing application had the highest priority). This measure somewhat limited the effectiveness and the usability of the proxy DLL approach in the more recent versions of Windows, especially due to the changed search path precedence and write protection of directories such as the Program Files directory.

Also note that such techniques to DLL injection work only assuming that the programs being discussed are running at an elevated privilege level to start with, which may not be true on modern systems e.g. if User Account Control (UAC) is enabled. In such cases, the actions of processes may be limited and write access to some file locations may be denied, which can limit the usability of proxy DLLs. It is, however, possible to instruct Windows to look in the application-local directory first by creating an appname.local file in the same directory as the executable (this requires write access to that directory which may be protected) or to specify a new DLL search path by calling *SetDllDirectory* in the target application.

The proxy DLL and similar approaches have been applied in the frameworks for detection of code injection attacks [13], DirectX function interception [15] and application isolation services [14].

(ii) *Modification of Windows registry*: A certain value in the Windows registry can be set to the path of a DLL[§], which is afterwards loaded into every application that provides a user interface (i.e. loads *user32.dll*) [16]. Although this approach is simple to implement and requires little programming knowledge, the specified DLL is loaded into most applications, which may be unnecessary. In addition, this approach cannot be used for applications without a graphical interface.

(iii) *Windows hooks*: are a mechanism provided by Windows for an application to monitor certain application or system-wide events (e.g. mouse or keyboard events). If a system-wide Windows hook is registered in a certain DLL, Windows will (for most hooks) load that DLL into all applications that trigger the hook (e.g. the application's user interface receives mouse movement or a keyboard event) [12, 17]. Although this approach is simple to implement and requires little programming knowledge, the specified DLL is loaded into all applications that trigger the hook, which may be unnecessary. In addition, it is not very flexible, since the DLL is loaded only after the application triggers the hook; hence, the time of injection cannot be explicitly imposed on the target application; some applications may not even trigger such hooks at all (e.g. applications without a GUI). For example, the WH_GETMESSAGE hook is triggered whenever a *GetMessage* or a *PeekMessage* function is called, which happens very often for GUI applications after they are initialized and enter the message loop.

[§] A value named 'AppInit_DLLs' under the 'HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows' key—modification requires administrative privileges.

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 3

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

(iv) *Using a debugger*: Using the debugger API, a thread of an application can be hijacked to execute arbitrary code (e.g. loading a DLL) by pausing it and rewriting its currently executing code. After the replacement code finishes executing, the original state is restored and the application is resumed [10]. This approach is reliable and provides greater control of the injection process. However, it does not work if the application is already being debugged or if the application prevents debuggers from interfacing with it. This approach is also somewhat more complex compared with those described above.

(v) *Patching the IAT*: When write access to the executable is allowed, one may also patch its Import Address Table (IAT) in the file itself to include the DLL that needs to be injected. This requires some knowledge of the PE format, fix-ups of the PE and its checksums. This approach allows early injections, but it has a major downside of having to change the executable on disk, which may not be possible (for example when User Account Control is enabled on modern operating systems) and could also break application-specific checksums or signatures. In contrast, patching applications in memory instead of on the hard drive usually bypasses checksum and signature checks. Such IAT patching is possible e.g. using PE modification software *CFF Explorer* [18].

(vi) *Reflective injection*: Since DLL injection can inflict security issues, some applications may try to defend themselves against it. For every process, Windows maintains a list of all loaded modules (such as the main program module and the DLL modules). If some DLL is injected from another process, Windows will register it in this list, making it possible for applications to detect such manipulation and act accordingly. The recently invented *reflective DLL injection* [19] makes detection of DLLs injected using this approach more difficult—e.g. the injected DLL cannot be discovered by using *EnumProcessModules* (from the PSAPI library). Reflective DLL injection is a low-level approach to DLL injection, involving the implementation of a minimal *PE loader*[¶] that injects the DLL in a stealthy way and remains hidden from the target application.

Alternative methods for detecting injections are by monitoring the instruction pointer [13] and monitoring the memory content [20].

2.2. API hooking

There are three major variations of hooking:

- *API hooking*, which hooks functions provided in a DLL,
- *function hooking*, intended for hooking arbitrary functions, and
- *code hooking*, which is the most general variation and allows redirection of arbitrary machine code.

Although the term *API hooking* represents only the first of the above variants, in practice it may also often denote any of the other two variants due to the capability of the latter two approaches to hook API functions, as well. In the following, we briefly describe the most popular approaches to implementation of API hooking.

(i) *Detours*. In 1999, Microsoft released its own API hooking library *Detours* [2], which is an implementation of function hooking (it partially supports code hooking, as well). Since the idea of our hooking method is based on this approach, we describe Detours to a greater extent in the following paragraphs.

The Detours library implements hooking by overwriting the start of a function's machine code with an unconditional jump instruction to a replacement function. Since the start of the original function has been altered, the library creates a *trampoline function*, which contains a copy of the overwritten code and an unconditional jump back to the unmodified part of the

[¶]The Windows program that is responsible for mapping executable code from disk into memory and performing initialization.

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 4

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

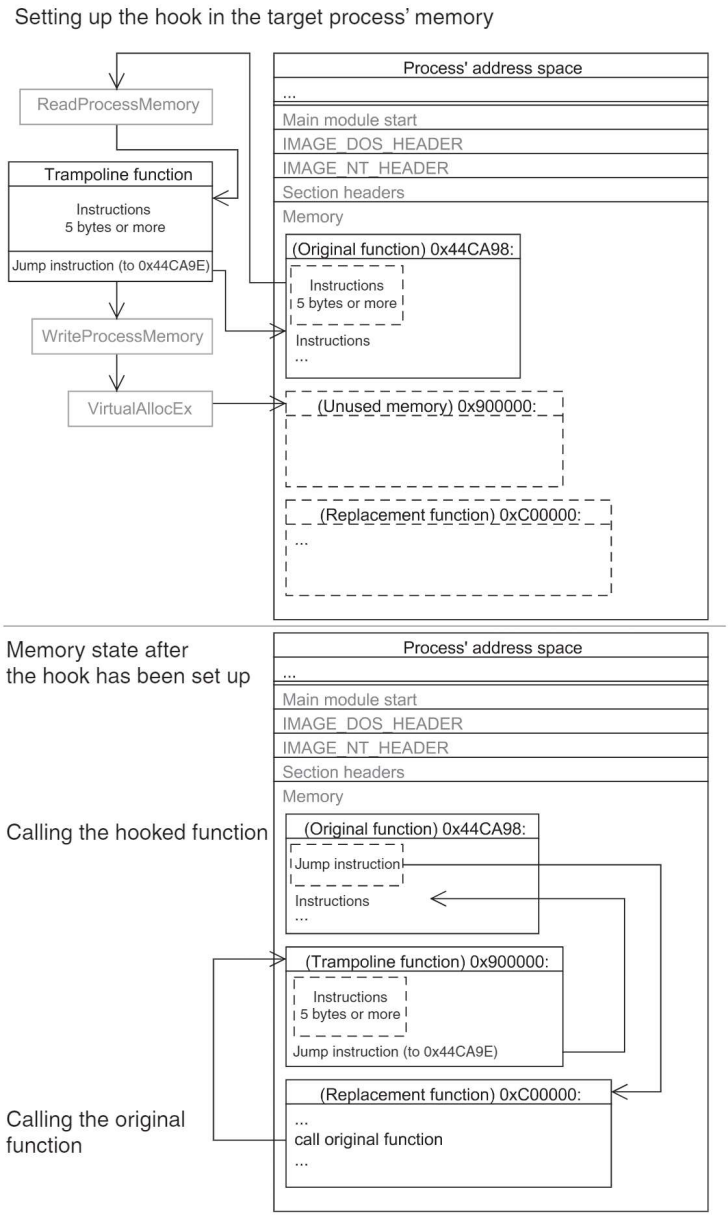


Figure 1. Detours hooking method with some technical details. *Above*: Creation of the trampoline function, which is created in the process' address-space. *Below*: The process after the applied change, and a call of the hooked function as follows. The original function contains an unconditional jump to the replacement function. The replacement function invokes the original function through the trampoline. The last instruction in the trampoline jumps to the unchanged part of the original function and proceeds with the execution of original code.

original function. In this way, the replacement function retains the possibility of executing the original function through the trampoline.

Figure 1 illustrates how Detours performs function hooks. The key part of the approach is in creating the trampoline function, which is performed as follows. First, a chunk of memory at the function's memory address is copied into a temporary memory buffer. Exactly 5 bytes need to be overwritten, but since the IA32 architecture uses variable-length instruction sizes, a chunk bigger than 5 bytes may be required (the chunk must be big enough so that any instruction that starts before or at the fifth byte is fully contained in the chunk). Then, the instructions are parsed from the memory chunk in the original function code until

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe
IPR2023-00895
KASPERSKY EX1030 Page 5

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

an instruction that starts after the fifth byte is reached. After the number of instructions contained in the first 5 bytes of the function's code is determined, these instructions are copied to the trampoline function. An unconditional jump instruction (JMP) is appended at the end of the trampoline, referring to the address of the first instruction of the function's code that was not copied to the trampoline.

At this point, if the trampoline's memory address was called, the result would be the same as calling the original function itself. However, the result of the above procedure is that the first 5 bytes of the original function can now be modified. To complete the hooking procedure, these 5 bytes are rewritten with a JMP instruction (which is exactly 5 bytes long—hence the need to copy at least 5 bytes into the trampoline). The address of the JMP instruction is set to the address of the replacement function. To reverse the hooking process (i.e. unhook a function), the cached instructions must be copied from the trampoline function back to the address of the original function, rewriting the previously written JMP instruction.

The advantage of *Detours hooking* is that it is fast and can be used to hook practically any code. However, in cases where the start of the code contains certain instructions (e.g. instructions that contain relative memory addresses), these instructions cannot be executed from a trampoline function, since the trampoline resides in a different memory location as the original program code. Therefore, Detours cannot be used in such cases without further modification (which is possible, but not implemented in Detours). Although such cases are rare, they still present an obstacle in the implementation of hooking. In the following section, we will present an approach that makes hooking of such functions possible, as well. It should be noted that in general, Detours could be extended to a point where it would be able to hook memory-relative instructions; however, handling such marginal cases would be particularly difficult. Our solution to this problem is not as complex and eliminates some problems that arise when using a trampoline (such as memory-relative instructions pointing into the trampoline itself). Our method also allows hooking of any single instructions and replacing them with arbitrary code, which is not possible with the Detours approach.

(ii) *Hooking by Import Address Table patching* (IAT patching [21–23]) is an even more widely known approach. Each PE executable (e.g. an EXE or a DLL) contains an *import table*, which holds information about DLL-provided functions that are used by the executable. The import table contains an IAT for each DLL that is included. The IATs are overwritten by the operating system with actual function memory addresses after loading the executable into memory. When the executable needs to call a function, it simply looks up the function's address in its own IAT. By patching the IAT, it is possible to redirect (hook) a function call to a different function than originally intended.

(iii) *Export table patching* [2] is similar to IAT patching, but patches *DLL export tables* (e.g. in the executable on disk, which may be write protected) instead of executable IATs. The DLL export tables contain addresses of all functions that the DLL provides. Patching either the executable IATs or DLL export tables would produce a similar result (the main difference being that such a hook is global compared with IAT hooking, i.e. affecting all applications that use the DLL).

(iv) *Hooking by Proxy DLL* [22]. In contrast to the above approaches, when injecting a DLL using the *Proxy DLL* method, it is already possible to perform API hooking without any further effort. The proxy DLL must have entry points of the same name. Besides the possibility of calling the original DLL to perform the operation, it may provide alternative implementations of these functions.

(v) *Hooking by the use of a debugger*. A debugger can position breakpoints in place of any function or instruction. When the breakpoint is triggered, the debugger redirects the flow of execution to another address, resulting in performing hooking. In our work, we extend this approach with our *Single Instruction Hooking* (SIH) approach by eliminating some of its downsides, such as the need for a debugger process, while still being able to hook any single instruction. Hooking by the use of a debugger has rarely been used, which can be

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 6

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

attributed to its bad performance. This kind of code redirection is used e.g. in *Immunity Debugger* [24, 25].

API hooking is a useful approach to application monitoring. Malware detection applications strongly rely on successful monitoring of the applications and some use API hooking to achieve this. Such examples include detecting shellcode injections [26], analyzing harmful API calls [27], application sandbox realization [28], and detection and prevention of kernel native API hooks [29].

Owing to the capabilities that DLL injection and API hooking provide, it is clear that they can be used either with honest or dishonest intentions. Making it possible to interfere with the predictable application behavior, they may be used either as the fundamental basis of anti-virus programs, malware detection programs, security tools and application extensions, or in viruses, rootkits or game cheats.

2.3. Windows API

The Windows API is a set of functions provided by the Windows operating system, which a programmer can use when developing applications for Windows. Windows API functions have been documented in several publications, the most known written by Custer and Solomon [30], Pietrek and coworkers [4, 5], Nebbett [6] and on the Microsoft Developer Network [31].

In our work, we heavily depended on several APIs:

- The debugger API, since it allows low-level interaction with applications [32],
- Memory management API [33],
- Security and threading API [34, 35],
- Portable Windows tool help API for process and thread enumeration [36],
- Windows NT/2000 Native API [6].

The APIs we have used are listed and briefly described in the Appendix.

3. ADVANCED APPROACH TO EXTENDING APPLICATIONS

In our work, we focus on bypassing the limitations of the existing methods: Most DLL injection methods namely share a problem of not being able to enforce running of the injected code at a particular point and time in the application. To run the injected code, the *CreateRemoteThread* method can be flexibly utilized, but it is not able to inject during the initialization stage of an application (a short period of time after the application is started). This is undesirable when the subsequent hooking of functions in the initialization stage of the application is necessary (as with APIs such as *Direct3DCreate9*, *CreateWindowEx* etc., which are called very early in the initialization stage). Injection by using a debugger can overcome this problem, since a debugger is able to attach before even the first instruction of the application is executed. However, since many applications block debuggers from interfacing with them, debugging of such applications becomes infeasible.

In our work we therefore use a mixed approach between the *debugger-based method* and a technique similar to the *Detours API hooking*. Although Detours has a limitation of not being able to hook all functions (functions that contain instructions in the first 5 bytes of code that use relative memory addresses), some other methods (e.g. IAT hooking) allow hooking of such functions. However, these methods have further limitations, such as not being capable of hooking functions that are not exported and not being able to hook arbitrary code, which may be required. Our goal is to provide a technique capable of hooking any code or function, while still allowing debugging of the application and producing low overhead.

In our approach, we implement application extensions as illustrated in Figure 2. The main idea is to use the DLL injection to introduce the new program functions and afterwards use API hooking to force the application to use them. The following Section 3.1 presents our novel approach to DLL injection and Section 3.2 our novel approach to API hooking. The C compatible

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 7

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973, By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



Figure 2. Extending a third-party application using DLL injection and API hooking. By using DLL injection we introduce our own (replacement) version of the original function and by using API hooking we force the application to use it instead of the original.

code of our approach is available at the address `http://zeta.mrbrdo.net/injection_hooking_08/`.

3.1. Debugger-aided DLL injection

As mentioned before, state-of-the-art DLL injection techniques utilize the *CreateRemoteThread* API to perform injection. This method is generally reliable; however, it cannot perform injection into an application that has not yet been fully initialized. As an example, suppose we create a new process that is placed in the suspended state before it is able to execute any code[‖]. If we attempt to inject a DLL into such a process, injection will fail, since the *CreateRemoteThread* API cannot handle processes that are not fully initialized. Since hooking can only be performed through a DLL loaded in the target process, not being able to inject the DLL represents an obstacle for this approach. However, many practical situations may require this problem to be addressed: for example, a developer may need to hook *CreateWindow* to change how the program's window will be created when the program is initialized, or hook *Direct3DCreate9* to manipulate a program that is using DirectX 9 for its Graphical User Interface (GUI).

In our work, we overcome these limitations by creating a hook of the process' entry point (the address of the first instruction in a process). The purpose of the hook is to perform injection and is done from an *outside process* (*injector process*), in contrast to Detours, which is designed to hook code in the same process. One way to perform this task would be by rewriting the entry point with a jump (JMP) instruction to code that loads a DLL (holding any code we may like to use, such as an API hooking engine and replacement functions), then rewrites the entry point back to the original state, and executes a JMP back to it, therefore proceeding with the original application behavior. However, determining the exact entry point address for uninitialized processes is not trivial in this case, as the operating system has not yet gathered all the information about such a process. Additionally, with this approach, memory protection issues may also arise, since the memory must be unprotected to allow rewriting of the entry point.

To avoid the described issues, we use the debugger API to manipulate the process' main thread context, which among other data also holds the address of the next instruction to be executed—the program counter (PC register or EIP register on Intel architecture) and is alterable from an outside process (note that it is only possible to safely modify the EIP of a suspended thread). The core idea behind the debugger-based injection is to pause the process (its main thread), rewrite its currently executing code and resume the execution. *Debugger-aided DLL injection* works by modifying the process as illustrated in Figure 3 and described in the following:

(i) Memory is allocated in the target process to hold the string with the path of the DLL and for instructions that load the DLL specified by the provided string. The DLL loading code must also ensure that registers and flags are unmodified after loading the DLL (for this purpose, instructions PUSHFD, PUSHAD, POPAD and POPFD are used) and must also signal the injector process that it has finished. We also append an unconditional jump to the address of the instruction being executed before the thread is suspended.

It is important at this point to discuss how to signal the injector process, which has to be signalled that the code has finished and to determine that the resources (memory) can be released. Usually, a mutex, an event or a semaphore could be used to signal the injector process, but since API calls are necessary for that, there are problems with such

[‖]This is performed using the *CreateProcess* API and passing *CREATE_SUSPENDED* as the *dwCreationFlags* parameter.

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 8

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

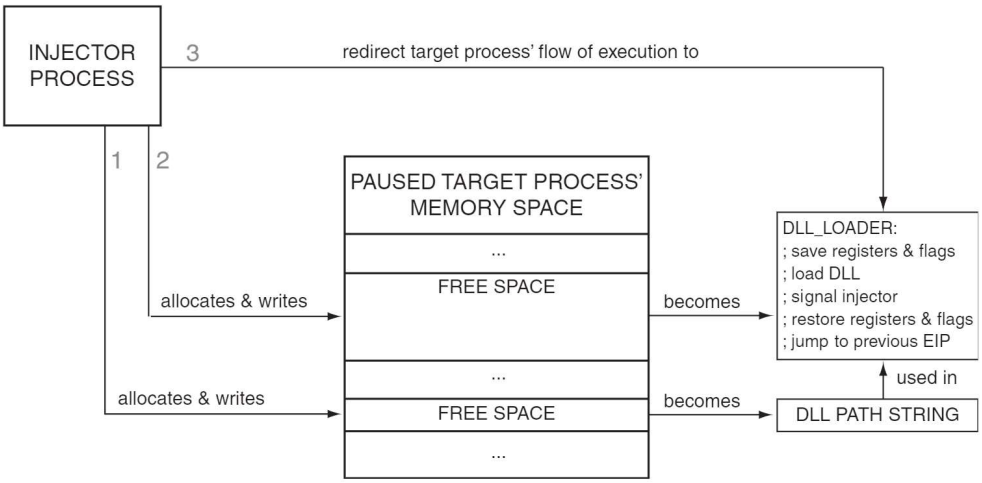


Figure 3. *Debugger-aided DLL injection*: An illustration of the first two steps of *Debugger-aided DLL injection*. First, the injector process writes a string with the path of the DLL to be injected into the target process' memory. Second, the injector process writes short low-level DLL loading code into the target process' memory, which uses the string written in the previous step. Last, the target process' flow of execution is redirected to the DLL loading code. After the DLL has been loaded, the part of the DLL loading code is overwritten so that the injector process can determine that the DLL had been loaded.

an approach due to the following. First, at the moment the event is signalled, the code has not yet finished execution (at that moment, the EIP would still point somewhere into that API's code). It is also not possible to only poll the EIP to determine if the code is still running, because APIs reside outside of our code snippet and must be used to load the DLL. Any approach of synchronization using APIs is unusable because of these problems. It would be possible to use an 'int 3' breakpoint instruction to signal the injector, but for that the injector would have to be debugging the target application, which is otherwise not necessary—therefore, it is beneficial to avoid using this technique (especially since some application may block debuggers, or may already be debugged).

Because of these problems, we have decided to use the approach, described in the following. First, note that after the DLL has been loaded (by using *LoadLibrary*), no other CALLs (e.g. to API functions) need to be made in the DLL loading code except the unconditional jump at the end of our code. Therefore, after the DLL is loaded, the EIP can be polled to determine if our code is still executing. To solve the issue of EIP pointing outside our code during the API call (therefore polling the EIP does not help us at that point), we can use a variable that we set after *LoadLibrary* has finished and no more CALLs will be made (writing such a variable can easily be accomplished by a single MOV instruction). Furthermore, because there is no risk of instructions already executed in our code snippet to be executed again, that memory can be reused instead of a separate variable if we have not protected that memory space against writing. By polling this memory location as well as the EIP, the injector process can determine whether the code has finished execution. Before the MOV instruction changes that memory location, it can be used as evidence that injection has not finished; later, the EIP can be polled because there are no more jumps until the end of the injection code. It must be noted, though, that while reading the target thread's context, the thread must first be suspended as this is the only way to safely read the context.

(ii) After we prepare the memory, we change the address of the next instruction (the EIP register) of the process's main thread to point to the memory location of our DLL-loading code**. When the thread is resumed, our code will therefore be executed.

**This is achieved using *GetThreadContext* API to retrieve the present context of the process' main thread and manipulate the context (i.e. change the EIP) using the *SetThreadContext* API.

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 9

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

(iii) During the execution of our DLL loading code, the injector process is polling the target's EIP and the memory location used for synchronization. After the injected DLL has been loaded by the target thread (in the previous step), the injector process can then safely release the memory allocated in the first step, which is not needed anymore.
(iv) Because our code snippet has jumped back to the instruction that was being executed prior to injection and all registers and flags have been preserved, the target thread now continues its normal execution. The target application's internal state is now identical as prior to the modifications, with the exception of the additional successfully loaded (injected) DLL. The application will therefore proceed as if it has not been modified or interrupted, whereas the DLL will be able to run in its context and perform modifications if needed.

The target application can later detect that a DLL has been injected into its context by enumerating its modules, e.g. by using *EnumProcessModules* from the PSAPI library (in that sense, this type of injection is not stealth). The benefit of our approach, being slightly more complicated than using *CreateRemoteThread*, is that it supports early injections. While this is also true for *reflective injection*, it is far more complicated and less reliable as it requires implementation of a *PE loader*, whereas our approach uses *LoadLibrary,* which is guaranteed to work on all versions of Windows (when the DLL exists and is valid).

3.2. Single instruction hooking

Having performed the DLL injection as described in the previous section, the next step is to ensure that the application uses the code in the injected DLL. In this section we describe an approach to hooking, which we term as SIH, which is similar to hooking by the use of a debugger and to some extent to Detours, as well (please refer to Section 2). SIH uses approaches common to debuggers, such as setting int 3 breakpoints and later intercepting their execution. The main difference between a debugger and SIH is that SIH does not require an outside debugger process to catch the traps raised by the int 3 instruction. SIH has the following main advantages over Detours:

- *it supports hooking of single instructions*: When parsing instructions, Detours determines if the processed instructions use relative memory addresses. If such an instruction is reached, hooking fails. However, the aim of hooking is to be able to hook any single instruction, even when the instruction is only one byte long, providing great freedom to a developer. This motivated us to base SIH on debugger-based hooking, but without using any external process (e.g. a debugger), while still allowing us to hook single instructions. In this way, we overcame this limitation in Detours,
- *it supports hooking of instructions that use relative memory addressing modes*: A very important advantage of using a debugger-based approach for hooking is the fact that we are able to change memory protection settings, allowing runtime changes to the application. This is the crucial fact that allows us to hook instructions referring to relative memory addresses, because we are able to execute those instructions at the memory location they originally resided at (by temporarily copying them there when needed). This is different compared with Detours, which permanently moves instructions to the trampoline and executes them from that location (which is different compared with their original location); hence, instructions using relative memory addressing will fail to work when executed from the trampoline. As a breakpoint is only 1 byte in length (compared with 5 bytes needed by a jump instruction), the amount of memory that needs to be changed is minimal; however, as it is apparent from the performance evaluation, exception handling is a time-consuming process.

An approach already exists in theory that is able to hook single instructions—the method we refer to as *hooking by the use of a debugger*, as seen used in Immunity Debugger [24, 25]. Because we are using the debugger API and debugger breakpoints, our approach is similar to how a debugger works by setting 'int 3' breakpoints and servicing them. However, one of the main and most important differences is that we do not use an outside debugger process to achieve hooking, which among other things slightly improves the performance and also works on the applications that

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 10

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

prevent debuggers from interfacing with them. In detail, *hooking by the use of a debugger* requires the use of a separate process (i.e. the debugger), which acts as the *control part* of the hooking system. The replacement code must reside in the target process (e.g. in an injected DLL, the *data part*). The code size of a separate debugger process can represent an overhead and is best avoided. In addition, resources must also be spent for the synchronization between the control and data parts, and inter-process communication is typically much slower than intra-process communication. An additional impractical issue is also that the usage of the debugger requires the presence of its executable file, which may be unwelcome in terms of implementation simplicity.

In the IA32 (PC) architecture, the only possibility to enable hooking of a single 1-byte-long instruction is using a debugger breakpoint. The debugger breakpoint itself is also a 1 byte instruction, and can therefore be used to replace any other instruction of an arbitrary length. However, for practical use, it is imperative to be able to service a breakpoint without the need for a separate process (i.e. a debugger program). Since a debugger breakpoint is implemented just as one type of an exception in Windows, the requirement for a separate debugger process can be avoided by the use of the Windows exception handling system. When an exception occurs in a program, Windows NT services it with an API function *KiUserExceptionDispatcher* (a user-mode function that is visible to the programmer and can be hooked like any other API function). This enables handling exceptions without using an external debugger program.

The clear advantage of having to replace only 1 byte compared with 5 bytes needed by Detours is that any instruction can safely be hooked. The 5 bytes overwritten by Detours may cause an application to malfunction when an address in-between these 5 bytes is the destination of a jump instruction. As a debugger breakpoint is only 1 byte in length, this issue does not exist with SIH. Note that in cases when an external debugger program is attached to the process, Windows circumvents the *KiUserExceptionDispatcher* function when exceptions occur. Nevertheless, in such cases it is still possible to catch exceptions by hooking other functions (i.e. the *RtlDispatchException* function). To hook the *KiUserExceptionDispatcher* API, any existing API hooking technique can be used (e.g. *Detours hooking*). Once we have this API hooked, we can catch exceptions in the current process, which means we can catch breakpoints in the program. When a breakpoint is encountered, Windows calls the *KiUserExceptionDispatcher* service program, which we have hooked.

SIH is schematically illustrated in Figure 4 and described in the following. A breakpoint (int 3) instruction is written over every instruction that needs to be hooked (or the first instruction of an API when hooking API functions). When the breakpoint is reached, the *KiUserExceptionDispatcher* exception servicing function will be invoked. The parameters provided to the function contain information such as the type of the exception and where the exception occurred. By storing the addresses of hooked instructions in an array or a similar structure, it is then possible to determine if the exception is the result of our hook breakpoint.

After we determine that an exception was generated by a breakpoint placed for the purpose of hooking, we can either redirect the flow of execution to a new address (i.e. replacement code) or allow original instructions to execute (e.g. when the replacement code needs to call the original code or function). The latter is slightly more complicated because the breakpoint instruction must be rewritten back to the original byte. This has a big time implication, since while the breakpoint is removed, all other threads in the process must be suspended to ensure that we do not miss any other thread trying to execute the hooked instruction. As we only need to wait for one instruction to be executed before we can write the breakpoint instruction back, we would like to execute only one instruction and immediately trigger another exception (thus triggering the *KiUserExceptionDispatcher* again). This can be accomplished by setting the TF bit (Trap Flag) in the EFLAGS register of the thread's context (retrieved by *GetThreadContext*). The TF flag causes a debug interrupt exception after the execution of a single instruction, and is the basis of single-stepping in a debugger. After the instruction is executed, our exception servicing function will be called again and we can restore the breakpoint byte, resume all threads and proceed with normal execution. Note that to speed up thread enumeration (for purposes of suspending and resuming them), one can keep a local copy of the list, and hook *ZwCreateThread* and *ZwTerminateThread* to catch the creation and destruction of all new threads.

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 11

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973, By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

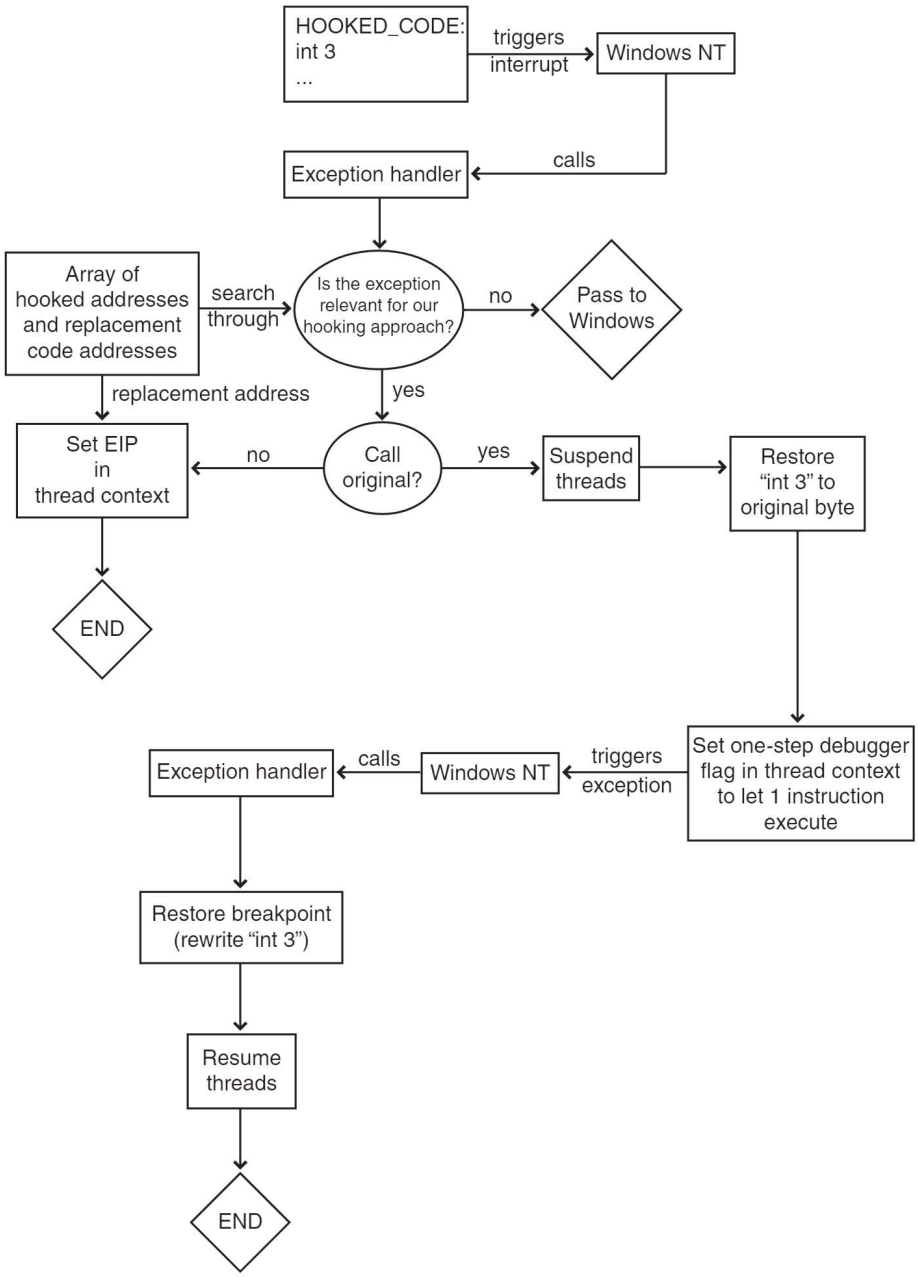


Figure 4. Single instruction hooking.

It is possible for an application to detect if SIH is being used by using several different techniques. Since SIH must patch (hook) *KiUserExceptionDispatcher*, one may verify that the beginning of this API is not modified. If it is, SIH may be in use and can be prevented from working if the modified part is restored. This would, however, break the application, since debugger breakpoint instructions would still be present and would throw an exception. The application could therefore detect SIH, display a message, and terminate. Other ways would be the application hooking of *KiUserExceptionDispatcher* and preventing SIH's handler to intercept breakpoints, or possibly scanning the memory for breakpoints or changes in general. It would also be possible to detect SIH by hooking *VirtualProtectEx* and seeing if memory containing code is unprotected against writing, since SIH relies on that API to be able to rewrite instructions with breakpoints. Of course,

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 12

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

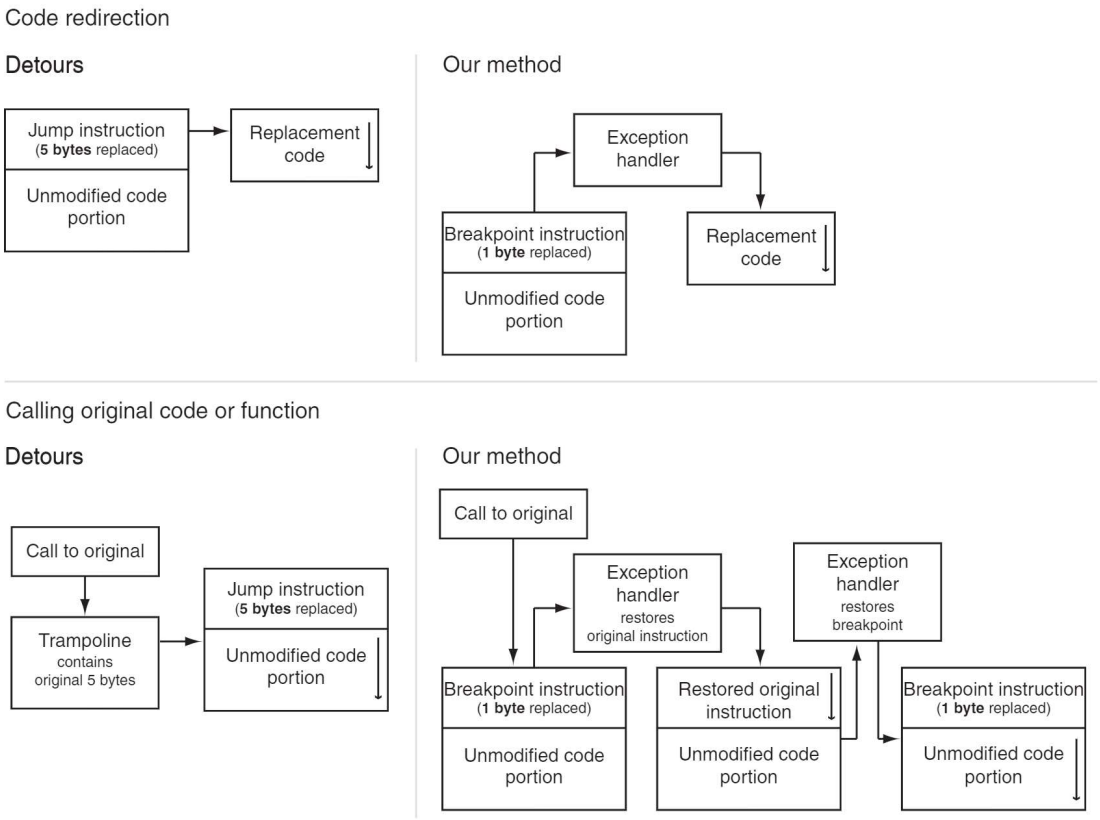


Figure 5. Comparison of Detours and SIH. *Above*: Comparison of how code redirection is performed. *Below*: Comparison of how original code is called.

SIH could be prevented to work by preventing DLL injection, because SIH needs to be used in the context of the target application.

Comparison of SIH and Detours: A simplified graphical comparison of Detours and SIH is given in Figure 5. The upper half of the figure illustrates how code redirection is performed in each approach: Detours uses a jump instruction to immediately redirect the flow of execution to the replacement function. SIH, on the other hand, uses a breakpoint to trigger an exception and subsequently the exception handler, which can then redirect the flow of execution to the appropriate replacement function. The bottom half of the figure shows how the original code can be called (e.g. from the replacement code): Detours requires the programmer to call a particular address (the trampoline) that executes cached instructions and proceeds to the unmodified code part after the jump instruction; SIH does not require calling of such particular addresses, but rather the exception handler is allowed to take care of rewriting the original instruction back to its original location, executing it and proceeding execution at the unmodified code part after restoring the breakpoint instruction.

4. PERFORMANCE EVALUATION

DLL injection: While speed is an important factor when comparing the performance of API hooking approaches, this is not the case with DLL injection due to the fact that injection needs to be performed only once in an application's lifetime and is considerably fast. While the speed of the *Debugger-aided DLL injection* is comparable to that of other techniques, it has the advantage of being able to inject into processes that are not fully initialized. When a process is already initialized (i.e. has been running for some time), our injection technique has no advantage over the existing

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 13

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Table I. Comparison of capabilities of **DLL injection** approaches.

Injection type	Windows registry	Windows hooks	Proxy DLL	Create remote thread	**Debugger-aided DLL inj.**
Applications with a GUI	√	√	√	√	√
Applications that use a DLL	×	×	√	√	√
Applications that do not use DLLs	×	×	×	√	√
Early injection is required	×	×	×	×	√

Table II. Comparison of capabilities of *API hooking* approaches.

Hook type	IAT hook	Detours	SIH
API or exported function	√	√	√
Any function	×	√	√
Instruction not using relative addressing	×	√	√
Instruction using relative addressing	×	×	√
Single byte instruction	×	×	√

ones; however, the need to inject into a process right after it has been started is very frequent and real. *Debugger-aided DLL injection* is compared with other commonly used approaches in Table I.

In our experimental evaluation, we compared the worst case injection times for the two most capable and thus most time-demanding approaches given in Table I. Those approaches are: the *CreateRemoteThread* approach and our approach—Debugger-aided DLL injection. Only to get a comparative impression of injection times for both methods††, we measured the execution time of *CreateRemoteThread* method (which was 1 ms) and the time of the Debugger-aided DLL injection (which was 17 ms). Note that the given times represent the averages of the repeated measurements and that they should be considered strictly informative, since the other processes and hardware latencies contributed to the measurements. Given that the injection is usually performed only once in an application's lifetime, both of the injection execution times can be treated as negligible. Despite the fact that the time measurements are in favor of an alternative approach, Debugger-aided DLL injection has an advantage that it enables injection also in cases when *CreateRemoteThread* does not. Since injection is a task that needs to be performed prior to the execution of the application and function calls, the greater time demand does not introduce any time implications for further application execution.

API hooking: *Single instruction hooking* enables hooking of arbitrary instructions and works in cases where other methods may fail, as well. The comparison of success in various scenarios is displayed in Table II.

The worst case execution times for approaches in Table II were: 0 ns for IAT hook, 1 ns for Detours and 1600 ns for SIH. Since the speed penalty of SIH is high, we use the Detours approach when possible, thus achieving the same speed with transparency to the programmer. When the Detours approach fails, however, SIH is used, which makes it possible to hook any API, function or instruction.

Since the tested approaches are general, they can be used in conjunction with an arbitrary application. The impact of the above execution times on an application depends on the application's complexity. Knowing the performance details of the target application, the developer will therefore be able to determine if the informative execution times provided represent a significant part of the total execution time and decide upon the acceptability of the selected approach, accordingly.

††Measured on an AMD Athlon(tm) 64 X2 Dual Core 5200+ 2.7 GHz processor with 2 GB of DDR2 RAM, running Windows 7 RTM.

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 14

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5. CONCLUSION

When programmers require modification of third-party applications, they frequently revert to DLL injection and API hooking to be able to apply modifications without modifying or having access to the application's source code. These two techniques are very often used in combination, as DLL injection allows the loading of an arbitrary DLL into an arbitrary application, which is a prerequisite for performing API hooking. We presented different DLL injection techniques (*CreateRemoteThread*, creating a proxy DLL, using the Windows registry or Windows hooks, injection using debugger utility functions and reflective injection). These techniques have different limitations, from not working in specific cases to being inconvenient for the programmer.

Having reviewed the API hooking techniques, we focused on the *Microsoft Detours* technique that is relatively fast. However, like the commonly used DLL injection approaches, the Detours technique has a limitation, as well: it is not able to hook certain instructions, which can sometimes even appear at the start of functions.

To avoid the aforementioned limitations, we developed new techniques for DLL injection and API hooking. We implemented DLL injection that works even when the application is not fully initialized. This enables subsequent hooking of initialization functions (which is required in DirectX, for example). To provide greater freedom in hooking, we also developed a hooking technique based on debugger utility functions that allows to hook arbitrary code. In these terms, the developed approach overcomes the limitations of the current methods.

We analyzed the performance of our techniques and compared them with the ones frequently used. We determined that *Debugger-aided DLL injection* allows injection in the wider set of scenarios, and its performance is similar to that of other techniques. We compared our API hooking technique with the common IAT hooking and Detours. Since SIH is based on Detours, the measurements showed that its speed is the same as that of Detours, where the Detours approach is applicable. In cases where the other techniques failed, SIH still worked with the cost of significantly increased time demands.

To conclude, the developed combination of DLL injection and API hooking approach provides more options for developers to extend and modify third-party applications, having equal or similar performance as current techniques, and work in cases where most or all current techniques fail. In these terms, the developed approach overcomes the limitations of the current methods, which can be in practice particularly useful in security analysis and reverse engineering applications. The ideas for future work include:

- extending the Detours approach to allow hooking of arbitrary instructions while maintaining its performance. This would include disassembling instructions and modifying them where needed, performing fix-up on relative memory addresses,
- hooking managed code (e.g. Java and .NET) and allowing unmanaged replacement functions (e.g. written in C++). This requires knowledge of both hooking managed and unmanaged code, and merging these approaches to allow seamless transition between managed and unmanaged codes,
- extending SIH by considering more efficient approaches to (or instead of) exception handling.

APPENDIX: RELATED WINDOWS API

In the following, the several Windows API functions that were used in our work are briefly described.

Debugger API functions, which allow low-level interaction with applications. The debugger API does not necessarily have to be used within a debugger, but an application must obtain the *SE_DEBUG_NAME* privilege to use most of the debugger API. To obtain this privilege, the following Windows API were used:

(i) *OpenProcessToken* opens the access token associated with a process. An access token contains the security information for a logon session. The token identifies the user, the

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 15

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

user's groups and the user's privileges. The system uses the token to control access to securable objects and to control the ability of the user to perform various system-related operations on the local computer,

(ii) *LookupPrivilegeValue* retrieves the locally unique identifier (LUID) used on a specified system to locally represent the specified privilege name. The privilege name of interest is *SE_DEBUG_NAME*,

(iii) *AdjustTokenPrivileges* enables or disables privileges in the specified access token,

(iv) *CloseHandle* closes an open object handle. This function should be used to close the process token opened by *OpenProcessToken*.

Windows API functions, which were used to perform DLL injection as well as API hooking:

(i) *OpenProcess* opens an existing local process object. The handle returned by this function is used as a parameter for other API functions. The handle must be closed by calling *CloseHandle*. To open a handle to another local process and obtain full access rights, the *SeDebugPrivilege* (named *SE_DEBUG_NAME*) privilege must be enabled. To perform DLL injection or API hooking, the full access rights are required,

(ii) *VirtualAllocEx* reserves or commits a region of memory within the virtual address-space of a specified process. To succeed, the process handle must have the *PROCESS_VM_OPERATION access right*,

(iii) *VirtualFreeEx* releases, decommits, or releases and decommits a region of memory within the virtual address-space of a specified process,

(iv) *VirtualProtectEx* changes the protection on a region of committed pages in the virtual address-space of a specified process. e.g., A part of memory may be set read-only—if writing to that part of memory is required, the protection must be changed to allow it,

(v) *ReadProcessMemory* reads data from an area of memory in a specified process. The handle must have *PROCESS_VM_READ access to the process*,

(vi) *WriteProcessMemory* writes data to an area of memory in a specified process. The handle must have *PROCESS_VM_WRITE* and *PROCESS_VM_OPERATION* access to the process,

(vii) *GetThreadContext* retrieves the context of the specified thread. The thread identified by the *hThread* parameter is typically being debugged, but the function can also operate when the thread is not being debugged. The thread context includes processor registers and debug registers (e.g. the *Extended Instruction Pointer* (EIP) register, which holds the memory address of the next instruction to execute),

(viii) *SetThreadContext* sets the context for the specified thread. This function can be used to change the thread's processor registers. The EIP register is of particular interest, as explained later,

(ix) *SuspendThread* suspends execution of the specified thread and the thread's suspend count is incremented. Suspending a thread causes the thread to stop executing user-mode (application) code,

(x) *ResumeThread* decrements a thread's suspend count. When the suspend count is decremented to zero, the execution of the thread is resumed,

(xi) *CreateProcess* creates a new process and its primary thread. Put differently, this function starts an executable file,

(xii) *CreateRemoteThread* creates a thread that runs in the virtual address-space of another process. This API is the backbone of the most popular DLL injection technique [1], but is not required in SIH. The *WaitForSingleObject* API is usually used to wait for the remote thread to finish. The process handle must have the *PROCESS_CREATE_THREAD*, *PROCESS_QUERY_INFORMATION*, *PROCESS_VM_OPERATION*, *PROCESS_VM_WRITE* and *PROCESS_VM_READ* access rights.

Portable Windows tool help APIs were used in our approach, which provide pausing or resuming all of the application's threads and enumerating through them:

(i) *CreateToolhelp32Snapshot* creates a snapshot of all threads currently running in the system,

(ii) *Thread32First* retrieves the first thread from the snapshot,

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 16

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

(iii) *Thread32Next* retrieves the next thread from the snapshot,
(iv) *GetCurrentProcessId* and *GetCurrentThreadId* APIs retrieve the PID of the current process and the current thread ID, respectively, allowing filtering by the Process ID (PID) and ignoring threads that are not running in our process.

REFERENCES

1. Richter J. Load your 32-bit DLL into another process's address space using INJLIB. *Microsoft Systems Journal* 1994; **9**(5):1–10.
2. Hunt G, Brubacher D. Detours: Binary interception of Win32 functions. *Proceedings of the Third USENIX Windows NT Symposium*, Seattle, WA, 1999; 135–143.
3. Keong C. Application programming interface. Available at: http://en.wikipedia.org/wiki/API [4 March 2010].
4. Alpaydin A, Maxey D, Pietrek M. *Undocumented Windows*. Addison-Wesley: Reading, MA, 1992.
5. Pietrek M. *Windows Internals*. Addison-Wesley Professional: Reading, MA, 1993.
6. Nebbett G. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing: New York, 2000.
7. Pietrek M. Peering inside the PE: A tour of the Win32 portable executable file format. *Microsoft Systems Journal* 1994; **9**(3):15–34.
8. Pietrek M. An in-depth look into the Win32 portable executable file format. *MSDN Magazine*, 2002.
9. Pietrek M. An in-depth look into the Win32 portable executable file format, part 2. *MSDN Magazine*, 2002.
10. Shewmaker J. Analyzing DLL injection. Available at: http://www.bluenotch.com/files/Shewmaker-DLL-Injection.pdf [4 March 2010].
11. Pulley R. Extending task manager with DLL injection. Available at: http://www.codeproject.com/KB/threads/taskex.aspx [4 March 2010].
12. Kuster R. Three ways to inject your code into another process. Available at: http://www.codeproject.com/KB/threads/taskex.aspx [4 March 2010].
13. Andersson S, Clark A, Mohay G, Schatz B, Zimmermann J. A framework for detecting network-based code injection attacks targeting Windows and UNIX. *Twenty-first Annual Computer Security Applications Conference* (*ACSAC'05*), Tucson, AZ, 2005; 49–58.
14. Lam LC, Yu Y, Chiueh TC. Secure mobile code execution service. *Twentieth Large Installation System Administration Conference* (*LISA 06*), Washington, DC, 2006; 5–5.
15. Koch M. Intercept calls to DirectX with a proxy DLL. Available at: http://www.codeguru.com/cpp/g-/directx/directx8/article.php/c11453 [4 March 2010].
16. Microsoft Help and Support. Working with the AppInit_DLLs registry value. Available at: http://support.microsoft.com/kb/197571 [4 March 2010].
17. Newcomer JM. Hooks and DLLs. Available at: http://www.codeproject.com/KB/DLL/hooks.aspx [4 March 2010].
18. NTCore. Explorer suite. Available at: http://www.ntcore.com/exsuite.php [4 March 2010].
19. Fewer S. Reflective DLL injection. Available at: http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf [4 March 2010].
20. Jang M, Kim H, Yun Y. Detection of DLL inserted by Windows malicious code. *IEEE International Conference on Convergence Information Technology*, 2007; 1059–1064.
21. Rowhani NR. Dll injection and function interception tutorial. Available at: http://www.codeproject.com/KB/DLL/DLL_Injection_tutorial.aspx [4 March 2010].
22. Ivanov I. API hooking revealed. Available at: http://www.codeproject.com/KB/system/hooksys.aspx [4 March 2010].
23. Pietrek M. Learn system-level Win32 coding techniques by writing an API spy program. *Microsoft System Journal* 1994; **9**:1–22.
24. Immunity I. Immunity debugger. Available at: http://www.immunityinc.com/products-immdbg.shtml [4 March 2010].
25. Damian Gomez I. Intelligent debugging for vulnerability analysis and exploit development. Available at: http://www.immunityinc.com/downloads/IntelligentDebugging.pdf [4 March 2010].
26. Sun HM, Lin YH, Wu MF. API monitoring system for defeating worms and exploits in MS-Windows system. *Information Security and Privacy*, Melbourne, Australia, 2006; 159–170.
27. Sathyanarayan VS, Kohli P, Bruhadeshwar B. Signature generation and detection of malware families. *Proceedings of the 13th Australasian Conference on Information Security and Privacy*, Wollongong, Australia, 2008; 336–349.
28. Willems C, Holz T, Freiling F. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy* 2007; **5**(2):32–39.
29. Keong C. Defeating kernel native API hookers by direct service dispatch table restoration. Available at: http://www.security.org.sg/code/SIG2_DefeatingNativeAPIHookers.pdf [4 March 2010].
30. Custer H, Solomon DA. *Inside Windows NT* (4th edn). Microsoft Press: Redmond, WA, U.S.A., 1998.
31. Network MD. Overview of the Windows API. Available at: http://msdn.microsoft.com/en-us/library/aa383723(VS.85).aspx [4 March 2010].

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

IPR2023-00895
KASPERSKY EX1030 Page 17

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

32. Kath R. The Win32 debugging application programming interface. Available at: http://msdn.microsoft.com/en-us/library/ms809754.aspx [4 March 2010].
33. Microsoft. Memory management. Available at: http://msdn.microsoft.com/en-us/library/aa366779(VS.85).aspx [4 March 2010].
34. Howard M, LeBlanc D. *Writing Secure Code* (2nd edn). Microsoft Press: Redmond, WA, 2002.
35. Microsoft. Processes and threads. Available at: http://msdn.microsoft.com/en-us/library/ms684841(VS.85).aspx [4 March 2010].
36. Microsoft. Tool help library. Available at: http://msdn.microsoft.com/en-us/library/ms686837(VS.85).aspx [4 March 2010].

1097024x, 2010, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.973. By Northwestern University Libraries, Wiley Online Library on [20/04/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Copyright 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/spe
IPR2023-00895
KASPERSKY EX1030 Page 18