Debugging Classification and Anti-Debugging Strategies

Shang Gao, Qian Lin, Mingyuan Xia, Miao Yu, Zhengwei Qi, Haibing Guan
Shanghai Jiao Tong University
Shanghai, P.R.China
{ chillygs, linqian, kenmark, superymk, qizhwei, hbguan } @ sjtu.edu.cn

Abstract—**Debugging, albeit useful for software development, is also a double-edge sword since it could also be exploited by malicious attackers. This paper analyzes the prevailing debuggers and classifies them into 4 categories based on the debugging mechanism. Furthermore, as an opposite, we list 13 typical anti-debugging strategies adopted in Windows. These methods intercept specific execution points which expose the diagnostic behavior of debuggers.**

Keywords - debugging, anti-debugging, software protection

I. INTRODUCTION

Debugging technique is a double-edge sword for software industry. It is usually employed as a powerful tool in software development to help the programmer find bugs or deficiencies. Or it is used by security engineer to analyze virus and malware. [2] However, hackers can adversely make good use of debugging to reconstruct the source code, which is also termed as reverse engineering. Hackers may debug certain application on public computer equipped with a generic debugger with stealth plug-ins. A hacker uses debugger to set a breakpoint at password input function, look up for correct buffer and wait for the debugger window to reveal the real password. Similar scenario may be applied to other private and sensitive account data acquirement.

Microsoft Windows implements series of APIs for developing custom debuggers for applications. It uses a callback method which allows the operating system to run the program by single-step. Mining INT3 instruction is another method for debuggers to pause and observe a process. Malware and some of the debuggers can also insert a structured exception handler to the running applications. When the debugged program is launched, it raises an exception and execution halt. There are various debugging methods and we propose a detail discussion in the next section. We classify them by different levers of privilege and assume different debug models.

As the requirement for software protection has gained general attention in the digital world, a great variety of copy-protection strategies of different forms have been developed to prevent cracking, tracing and reverse engineering. Majority of these mechanisms provide a reasonable level of security against static-only analysis. Current existing anti-debugging methods can be sorted into two categories. One is checking hardware or software debug registers for special value that debuggers may use to place breakpoints on processes. Another method is to detect user operation such as unexpected pause in execution.

The contribution of this paper is a theoretical one. We analyze the prevailing debuggers and classify them into 4 categories based on the debugging mechanism. Furthermore, as an opposite, we list 13 typical anti-debugging strategies adopted in Windows. These methods intercept specific execution points which expose the diagnostic behavior of debuggers. We believe that the cross-usage of these anti-debugging strategies makes a sound way to defend attacks based on debugging, i.e. software protection through anti-debugging is available.

The rest of the paper is organized as follows. Section 2 analyzes the popular debuggers and makes the classification. Section 3 describes the debugging mechanism in Windows OS. Section 4 introduces thirteen typical anti-debugging strategies adopted in Windows. We conclude the paper in Section 5.

II. PREVAILING DEBUGGER TYPES

In this section, we describe several typical debugging models utilized in most debuggers to generalize the features and requirements of debugging.

Debugging is the analysis of computer program which is performed by executing programs built from that software system on a real or virtual processor. Debugging medium could be diverse [6]. We classify all prevailing debuggers into four categories: user-mode debugger, kernel debugger, system-level debugger and simulator debugger. This classification stands on the difference in debugging mechanisms. As the computer system has evolved to such a complex architecture, debugger developers make use of every accessible resources, both hardware and software, to implement their effective debugging mechanism.

A. *User-mode debugger*

The user-mode debugger deploys APIs, offered by the operating system, to play the debugging tricks. When the debugger process starting up, it attaches itself to the target process or constructs debug environment by forking the debuggee process from inside. In order to trace the target, debugger uses some wrapped APIs to set breakpoints in the target process and analyze code. As long as the process hits to the breakpoint, a breakpoint exception will be triggered. After the exception returned, operating system kernel still

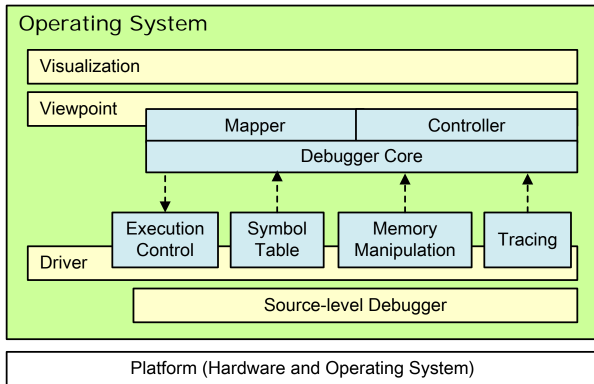

Figure 1 Kernel Debugger Model.

takes over the control and keeps the target process suspending. Further, it packages the relevant context and communicates with application debugger, then transfers the control flow to the debugger for post-processing. Afterwards, with respect to user's determination, the debugger will guild the kernel how to deal with the target process, such as stepping, resuming and canceling.

The representative debuggers of this category include Phantom, OllyDbg and OllyICE.

B. *Kernel debugger*

Kernel debugger occupies more privileged level than user-mode debugger to achieve more powerful debugging ability. The main components usually stay in kernel space of the operating system such as Microsoft Windows so that they are entitled to exploit kernel resource to rule debugging procedure. WinDbg and KD are typical members in this class.

As shown in Figure 1, the kernel debugger model is divided into several layers. Each layer furnishes different level of abstraction of the debugging procedure. Details on each layer are listed as following.

1) *Source-level debugger*

The starting point of the kernel debugger layered model is the source-level debugger that executes on the target platform. For the purpose of behaving as privileged as operating system kernel, source-level debugger always resides in the kernel space and holds the highest execution permission, i.e. Ring 0 level of Intel CPU. As the ground layer of the debugger, source-level debugger is of dependency, inherent with instruction analysis and debugging control flow. Different target architectures include or support different source-level debuggers that allow extraction of runtime information over proprietary interfaces. The driver layer unifies these source-level debuggers as far as possible to provide various services for debugging procedure.

2) *Driver Components*

Most of debugging functions and libraries exist as driver components. Drivers generally implement only a narrow set of the interfaces, restricting the capabilities that the debugging system can offer to the higher levels. Common interfaces include execution control (starting, stopping, resetting, and stepping), access to symbol data (memory addresses of methods), and memory manipulation (scanning and writing). Hardware-based debuggers also offer an interface, by default, for acquiring and reading real-time trace data from the target. For driver usually stays in kernel space, it share the common privilege level with OS kernel. Citing the functionality of execution control component as an example, it can respond to the application run-control request from upper level, as pausing the target process and switching the execution flow to debugger space.

3) *Viewpoint*

Viewed from the topmost layer, the model offers the user different viewpoints of the debugger. Now that applications are becoming so complicate that bare breakpoint and step forward tracing are not adequate for developers. When debugging a program, user often expects more information from different viewpoints, especially with object-oriented techniques. A viewpoint defines a specific view of the system and mediates between the target system and the visualization layer. The complete set of viewpoints offers insight to all aspects of the running program.

4) *Debugger Core*

Between viewpoint and driver is the debugger core composed of mapper and controller that bridge the layers. The viewpoint uses the controller to control the system from the model's viewpoint. Since source-level debugger has masked the platform difference, debugger core is platform independent except for some essential libraries and APIs. An execution step in a state chart corresponds to a complex sequence of commands at the source-level. The mapper updates the runtime information model that provides the basis for the visualization with information from the source-level debug drivers.

5) *Visualization*

Most current mainstream debugging engines provide console-based command line interfaces. A well-designed visualization interface plays an essential assistant role for debugger user. Debugger front-ends are popular extensions to debugger engines that provide IDE integration, program animation and visualization features, which are implemented on the visualization layer. Besides the interface design, the capability of this layer mainly depends on viewpoint functions supported by the beneath layer.

C. *System-level debugger*

System-level debugger is of operating system independency. It is designed to run underneath operating system so that the operating system is unaware of its presence. Unlike user-mode debugger and kernel debugger, system-level debugger exhibits a strong ability to suspend all operations in the target operating system when instructed. The notable delegate in this category is SoftICE. Due to its low-level capabilities, SoftICE is frequently utilized as driver debug-

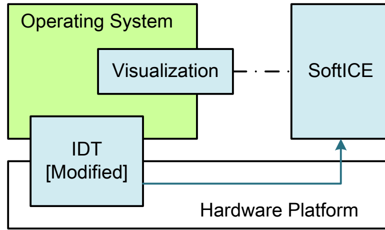

Figure 2 System-level Debugger Model (SoftICE).

ging tool; also, it is popular as a software cracking tool. Syser is another newcomer in this class.

Figure 2 presents the system-level debugger model, taking SoftICE as example. SoftICE replaces the original interruption disposal routines with its own version to obtain system control power. During the installation of SoftICE, it modifies the Interrupt Descriptor Table (IDT) and redirects some interrupt handlers, such as ``INT 3h" (breakpoint exception) and ``INT 31h" (8042 keyboard controller interruption), so that it could hook the relevant interruption or exception event for debugging usage. Most of its functionalities are done through I/O port manipulation (read/write), seldom using APIs. The implementation of visualization component is simply refreshing display mapping buffer in memory. Visualization interface will pop up if user presses the corresponding hot key. Some external functions such as log may need the kernel resource.

D. *Simulator debugger*

Figure 3 illustrates the model of simulator debugger. The significant feature of this category is that the simulator manager deploys simulation techniques, such as hardware emulation and software virtualization, to build up virtual machine container. The target applications or even operating systems are present in such container and run normally. The containers are actually some memory blocks and provide ``hardware" resource via simulation. As one of the component parts of the simulator, simulator debugger is convenient to debug programs within the virtual machine container because it owns the even higher privilege level than kernel (in the container). Bochs with internal debugger and ROR are the representative simulator debuggers.

III. DEBUGGING MECHANISM IN WINDOWS

Being one of the most prevailing desktop operating system, Microsoft Windows has been being widely used. Windows abstracts its own debugging mechanism from the CPU debug support facilities [7]. When exception occurred, CPU preserves all registers, constructs the TrapFrame in kernel stack and transfers the control flow to the corresponding trap disposal routine. Instead of dealing with the exception themselves, most disposal routines call API function CommonDispatchException() to build up ExceptionRecord and ExceptionFrame to prepare for exception dispatch. The ExceptionRecord records the exception code, exception address

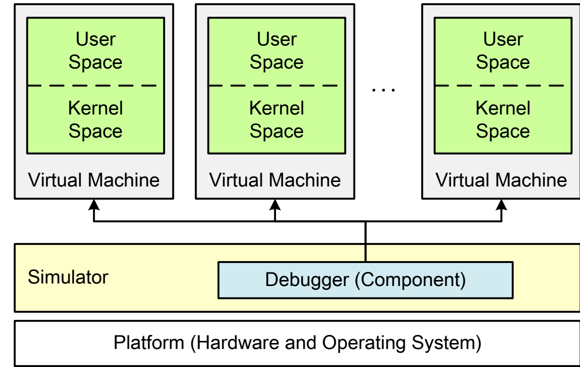

Figure 3 Simulator Debugger Model.

and some relevant parameters. Then, KiDispatchException() will be invoked to dispatch the exception. KiDispatchException() is the core API for exception disposal in Windows, managing exception dispatching.

Figure 4 shows the exception processing flow with KiDispatchException(). Interruption may occur in kernel mode or user mode, both of whose processing are similar except for some detail. When the interrupt occurs in kernel mode, as shown in Figure 4(a), the key disposal points are listed as following:

- The debugger has two chances to take over the exception processing by calling API KiDebugRoutine().
- Between these two chances, kernel will call RtlDispatchException() to search Structured Exception Handling (SEH) in kernel stack.
- As long as the exception is handled in one of the three steps above, the TrapFrame will be configured and the execution flow will return (using *iret* instruction) to interrupted point.
- Otherwise, if no handling can take over the exception, KeBugCheckEx() will be invoked to call forth blue screen of death (BSOD). In other words, the system will generate a fatal fault if the exception lacks the corresponding handling.

When the interrupt happens in user mode, the corresponding processing also offers the debugger two chances to handle the exception, as shown in Figure 4(b). The key disposal points are listed as following:

- The debugger exchanges debugging message though debugger port.
- In the first chance, if the target process has a debugger port, KiDebugRoutine() will send a message to it and wait for a reply.
- Then RtlDispatchException() of user mode version will search SEH in user stack. If the debugger handles the exception, the target process continues execution.

(a) Kernel Mode

(b) User Mode

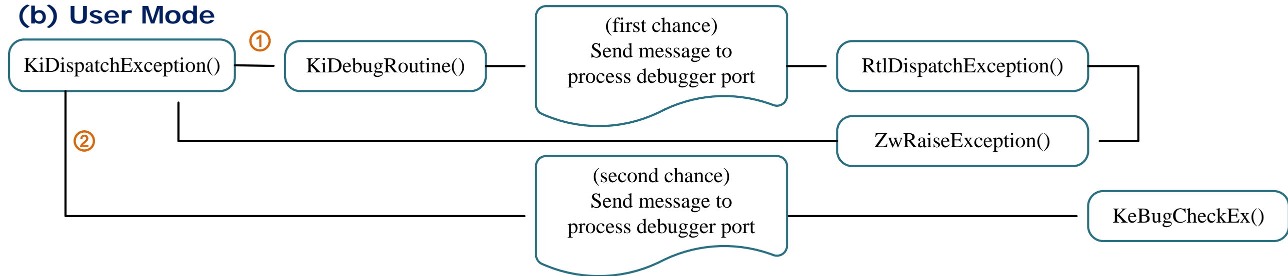

Figure 4 Exception processing flow with API KiDispatchException(). Interruption may occur in (a) kernel mode or (b) user mode. Both exception disposal flow fall into KiDispatchException(), and the processing are similar. Debugging could branch off within the procedure, and the debugger has two chances to take over the exception processing.

- Otherwise, the ZwRaiseException() system service will call KiDispatchException() a second time to process the exception. In the second chance, KiDispatchException() checks the debugger port directly to see whether there exists debugger to handling the exception this time.
- If the exception is sill unhandled within this routine, as that of kernel mode, BSOD is its story end.

IV. ANTI-DEBUGGING STRATEGIES

Thirteen anti-debugging methods are selected involving most of general anti-debugging tricks against debuggers on Windows, and grouped into three categories.

A. API

The most straightforward way to detect the presence or the operation of a debugger is to use Windows provided functions, which can be either documented or undocumented and exported in various .DLL and .SYS files.

- **Invoke *IsDebuggerPresent()*.** It indicates whether the calling process is being debugged by a user-mode debugger.
- **Check remote debugger.** *CheckRemoteDebuggerPresent()* is used to determine whether the specified process is being debugged. It can indicate the debugger resides in a separate and parallel process.
- **Test *SeDebugPrivilege*.** A process will have full control on CSRSS.EXE once it gets the SeDebugPrivilege privilege, which will be inherited to its child processes. If one process owns such privilege, it is probably under debugging.
- **Check parent process.** A user usually executes an application by clicking the shortcut or by command-line, so the parent process of the application should be either EXPLORER.EXE or CMD.EXE on Windows. Thus, if the parent process is something else, the application is possibly running in a debugging environment.
- **Find debugger window.** A running debugger can be found by iterating through all window handles. However, this anti-debug trick can only point out the existence of debugger program. It cannot indicate whether the specified process is being debugged.
- **Detect *CloseHandle()* exception.** Passing in an invalid handle to the CloseHandle() function will trigger an EXCEPTION_INVALID_HANDLE exception in the case of current process debugging. However, this will not happen in normal execution, which will return an error code as result.
- ***OutputDebugString()*'s *LastError*.** If the current process is running under a debugger, calling GetLastError() function next to OutputDebugString() function will get 0 as the return value, but it is not true without a debugger.

B. Special structure

Another way to detect debugger is to traverse and analyze Windows kernel structure. According to Windows System Error Handling mechanisms, when a process is debugging, certain structures are modified to enable dispatching debug messages to the debugger.

- **Check *BeingDebug* flag in PEB.** The BeingDebug flag in PEB is set when the current process is under debugging. Thus this flag can indicate the existence of debugger.
- **Check debugger port.** In Windows, the process debug port is set to the Windows subsystem's general function port when debugging the current process,

which ensures that related debug events route to the process's debugger.

- **Check debug object handle.** Debug object plays the same role as debug port so that checking debug object handle can identify whether the current process is debugging.

C. Exception

In the case that a process is debugging, the debugger will overtake the exceptions of debuggee process. Furthermore, the debugger is not likely to hand these exceptions back to the debugging process. Thus, by building a well-designed exception trap, a process is able to determine whether it is debugging or not.

- **Detect *INT3* exception.** INT3 instruction, sometimes referred to as a breakpoint exception, which can cause a CPU trap to occur in the operating system, provides a method for debuggers. If the process is running within a debugger, the debugger captures the INT3 interrupt and never hands it back. Thus, this instruction flow change can be view as a flag of debugging behavior.
- **Detect Single step exception.** A running process can set TF in EFLAGS/RFLAGS register to enable single step trap, which can be seen as a sign of debugging.
- **Detect *INT2D* exception.** The trap handler for ``INT2D" constructs an EXCEPTION_RECORD structure with an exception code of STATUS_BREAKPOINT, and then hands it over to the kernel debugger.

V. CONCLUSION

Debugging is widely used for runtime observation and manipulation of application program. This method is useful and helpful to software development, but it may also be wicked if malicious code or attackers employ it. This paper analyzes the prevailing debuggers and classifies them into 4 categories based on the debugging mechanism. Furthermore, as a opposite, we list 13 typical anti-debugging strategies adopted in Windows. These methods intercept specific execution points which expose the diagnostic behavior of debuggers.

[1] Lee Byeongcheol, Hirzel Martin, Grimm Robert, and McKinley Kathryn S, “Debug all your code: portable mixed-environment debugging,” In OOPSLA ’09: Proceedings of the 24rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, New York, NY, USA, 2009. ACM.

[2] Michael N. Gagnon, Stephen Taylor, and Anup K. Ghosh, “Software protection through anti-debugging,” IEEE Security and Privacy, 5(3):82–84, 2007.

[3] Philipp Graf and Klaus D. Muller-Glaser, “Gaining insight into executable models during runtime: Architecture and mappings,” IEEE Distributed Systems Online, 8(3):1–11, 2007.

[4] Kris Kaspersky, “Hacker Debugging Uncovered,” Independent Pub Group, 2005.

[5] Samuel T. King, George W. Dunlap, and Peter M. Chen, “Debugging operating systems with timetraveling virtual machines,” In ATC ’05: USENIX 2005 Annual Technical Conference on Annual Technical Conference, pages 1–15, Berkeley, CA, USA, 2005. USENIX Association.

[6] M. Russinovich and D. Solomon, “Windows Internals, Fourth Edition,” Microsoft Press, 2005.

[7] Chen Xu, Jonathon Andersen, Zhuoqing Morley Mao, Michael Bailey, and Jose Nazario, “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware,” In DSN ’08: The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 177–186, 2008.