March 23, 2015

THIRD PARTY FUNCTIONING SANDBOXING

Chris Sharp

Robert Shield

THIRD PARTY FUNCTIONING SANDBOXING

ABSTRACT

The subject technology dynamically patches functions exported and/or used by third party code to catch exceptions generated by third party code, to prevent the third party code from crashing other processes in a system. All exceptions that the third party code generates (e.g., either directly or indirectly) are handled before being propagated up to the system level, and before the system kills the process due to an unhanded exception.

PROBLEM

Third party code may be injected into virtual memory, and interact with the operating system and its executing applications. Sometimes this code is not well written and may cause crashes. If the code is unstable, and/or not useful for a user (like malware or adware), it may be desirable to prevent the code from being loaded in the first place. Previous solutions include blacklisting known malicious programs from being loaded by the system, and sandboxing programs to prevent exceptions thrown by the programs to affect other programs executed by the system. However, there is an abundance of third party code that users routinely download and execute (e.g., Silverlight which allows them to watch Netflix), and determining which code may cause problems is not practical. Additionally, blocking all unknown code (e.g., to increase stability) may create an unhappy user base. Thus, solutions are required to allow unknown code to execute without crashing the system.

DYNAMICALLY WRAPPING SYSTEM FUNCTIONS

When a process first begins, the executable file corresponding to the process is mapped into a virtual address space allocated for the process. The virtual address space may include multiple different components. For example, additional libraries may be mapped into the address space in order for the process to be able to call functions provided by them.

The nature of operating systems require processes to use certain system-level functions provided by the operating system to perform memory operations, such as loading libraries and/or allocated objects into its virtual address space. In this regard, before a process may write to the virtual address space the memory must be set to a write state, and before code in the memory space can be executed it must be set to an executable state. For example, the process may call a

system function (e.g., VirtualProtect) to change a protection level assigned to a region of the virtual address space of the calling process.

Because most operating systems allow processes to perform dynamic patching, all places that might be called into, for example, a third party dynamically-linked library (DLL) may be unknown. The list of functions exported by the library, however, provide (in most cases) a fairly good coverage of where exceptions could be raised from the third party code. The subject technology includes system-level code that dynamically enumerates and patches exported functions of code loaded into the virtual address space of an executing process. Patching may occur when the program embodying the process is first loaded into memory. For example, system functions responsible for loading the process into memory may include code for determining whether the process is one that would require patching, and patching the libraries used by the process (e.g., libraries that include system functions loaded into the virtual address space). When those functions are called to load the process, the patching is initiated. The patching may be effectuated by way of virtual wrappers which intercept calls to functions exported by in-memory objects.

Copies of system-level functions are typically provided by dynamic linked libraries of the kernel. When they are loaded into the virtual address space of the executing process, they are modified by the subject technology with custom wrappers. (See Fig. 1.) When the process later calls the system functions, the calls must go through the wrappers. In this manner, each call may be inspected by the operating system and a decision made as to whether the call should be allowed to complete as expected, or whether the call should be modified or blocked. In some implementations, the wrappers are configured to render the call ineffective while simply returning an expected response to the calling process. For example, the calling process may call a system-level function to modify or otherwise patch a portion of code in its virtual address space. The wrapper intercepts the call, prevents the code modification, and reports the modification as being successful to the calling process.

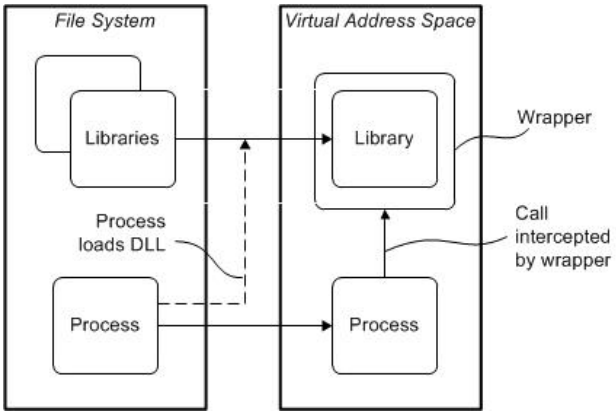

Fig. 1

In some implementations, the subject technology patches a DLL on detecting the loading of the DLL. When the process that loaded the DLL attempts to call functions of the DLL the function calls are intercepted by the wrapper and rendered useless. Accordingly, if the DLL was loaded with malicious intent, for example, to alter the normal operation of the system, the DLL is rendered inoperable and normal operation of the system is maintained. In some aspects, the subject technology requires little knowledge of what third party DLLs are loaded, since it merely examines the public (and local) functions of a library or process and exports and wraps them. In this manner, exceptions are prevented from propagating up to a point where the system is compromised. A potential alternative way to prevent third party crashes may include converting all exported functions into null operations. In this case, the third party code wouldn't generate exceptions, but the user may lose the functionality that the third party code was offering.

Using the subject technology, software is allowed to inject modules, including unknown code, into a system. However, when unwanted code is injected into the system, the unwanted code is rendered useless and stability of the system is maintained. Meanwhile, the process that injected the unwanted code operates as it normally does. In this regards, the subject technology allows third party code to run within a program, but greatly decreases the risk of code crashing the program. Accordingly, users are able to install whatever third party code they want, and the rate of crashes resulting from unstable or malicious code is decreased.