Using Kernel Type Graphs to Detect Dummy Structures

Brendan Dolan-Gavitt   Patrick Traynor

December 8, 2008

An open problem for signature-based scanners for kernel data structures in memory is the potential for attackers to create dummy objects that, while syntactically valid, are not actually used by the operating system. These fake structures can cause false positives when searching for data structures in memory, creating noise in which an attacker can hide malicious objects. As a result, a method for weeding out such false positives, separating the active instances from the maliciously placed dummies, is needed if context-free scanners are to be successful.

To solve the problem, we first noted that in most common operating systems, legitimate kernel data structures are highly interlinked: each structure generally contains one or more pointers to other kernel objects. Thus the structures in memory can be seen as a graph, where each structure is a node and an edge is formed when there is a pointer from one object to the next. This mini-project sought to investigate the use of graph-theoretic properties to distinguish between active and inactive kernel data structures. Our hypothesis was that inactive structures would have a very low indegree; as the structure is not currently used by the operating system, other kernel data structures should not point to it.

In order to generate a representation of such a graph, accurate information on all kernel data structures and their relationships is needed. We decided to work with the Windows kernel, as it was most familiar to the authors. Windows itself is closed source, however, most of its data types are freely available in the debug symbols (PDB files) distributed by Microsoft [3]. These types can be extracted using the open-source tool PDBparse [2]. The type data by itself serves as a template for the graph of actual instances in memory.

There are a number of complications that prevent the type data from being used directly. First, linked lists are heavily used throughout the kernel; however, the list pointers themselves are contained in a `LIST_ENTRY` object embedded in list each member, and they point not at the next member, but at the next `LIST_ENTRY` inside that member. We worked around this by manually annotating the raw type data with information on what type of object each embedded `LIST_ENTRY` pointed to, using information derived by hand from the Windows Research Kernel [4] source code.

Second, many data structures contain unions, and hence the type pointed to may be ambiguous; we opted to traverse each member of the union, rather than trying to guess the correct type. Our assumption is that attempting to traverse the graph from the incorrect type will eventually fail, as the pointers going outward will not point at valid memory. Finally, there are a number of structures that have pointers to void as members; the actual type pointed to in this case is not known, and we have opted to ignore them for now.

To construct the instance graph, we decided to write a new plugin for Volatility [5], an open source memory analysis framework. Volatility was chosen mainly because PDBparse has the ability to export data structure definitions in Volatility's native structure definition format. The plugin takes as input the virtual address of a seed node, along with the type of that node. For our experiments we focused on `EPROCESS` data structures (used in Windows to hold information about a running process), as attackers commonly attempt to hide these structures from security tools. The plugin traverses the instance graph using breadth-first search (implemented with a queue), and at each node gathers all valid pointers to other structures in memory, adding them to the queue. Pointers that do not point to valid memory, or that point into userspace, are discarded. The traversal terminates after a user-specified number of queue entries have been processed.

We tested the hypothesis that indegree would serve as an accurate way to weed out false positives by taking the memory from a paused VMWare snapshot and scanning its virtual address space using Volatility's built-in signature-based scanner. This gave us a list of candidate processes, which we manually classified into active and inactive based on whether or not they appeared in the main linked list of active processes. We then calculated the indegree of each candidate; the results are shown in figure 1. Unfortunately, based on these results, the active and inactive processes are not cleanly separable using indegree as a metric.

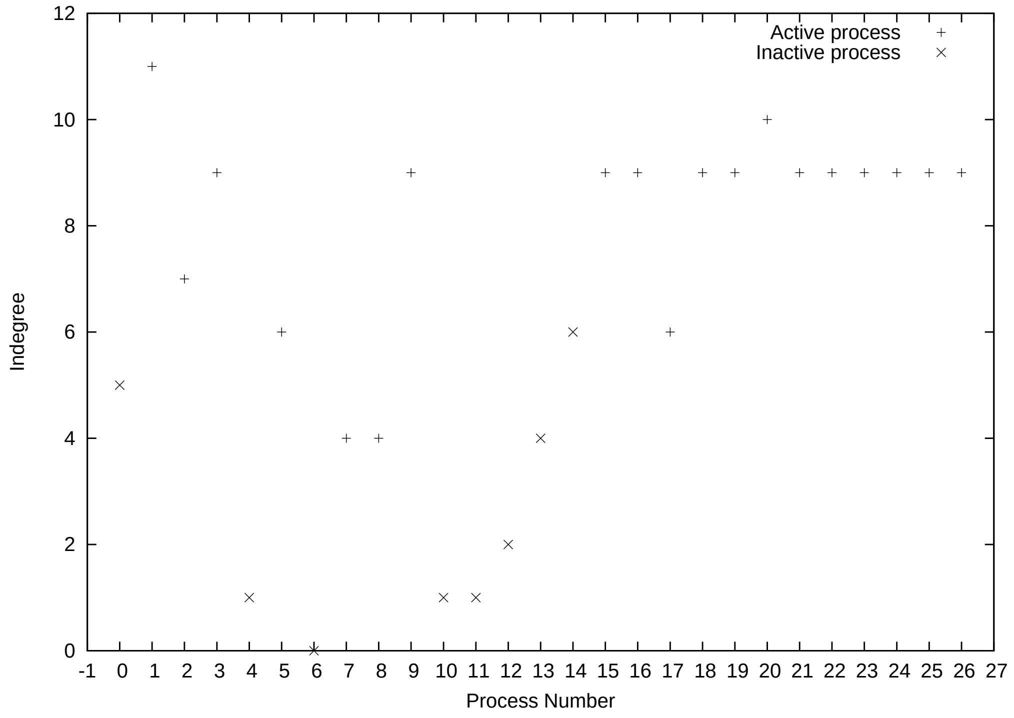

Figure 1: Indegrees of process data structures in a test memory image. No clear separation can be seen between active and inactive structures.

By itself, it appears that indegree is too sensitive to be used as an indicator of whether the structure is in use by the OS. This is because it is possible to have a high indegree with only a few supporting structures that point to the inactive structure; for example, one process (numbered 14 in the figure) had recently exited and was no longer active, but had six threads associated with it, which each pointed back to the now-inactive process. Thus, although neither the process nor the threads were still running, the process had an indegree of 6.

To counteract this effect, we sought an algorithm that would take into account not only how many structures pointed to the seed node, but also how many structures pointed to each of *those*, and so on. In other words, we wish to know not only how many structures point to our candidate, but how "important" each one is. This metric is, in fact, computed by the PageRank® algorithm [1]. By applying the PageRank algorithm to the graph of kernel data structures, we found that it gave a clear separation between active and inactive processes, as shown in figure 2.

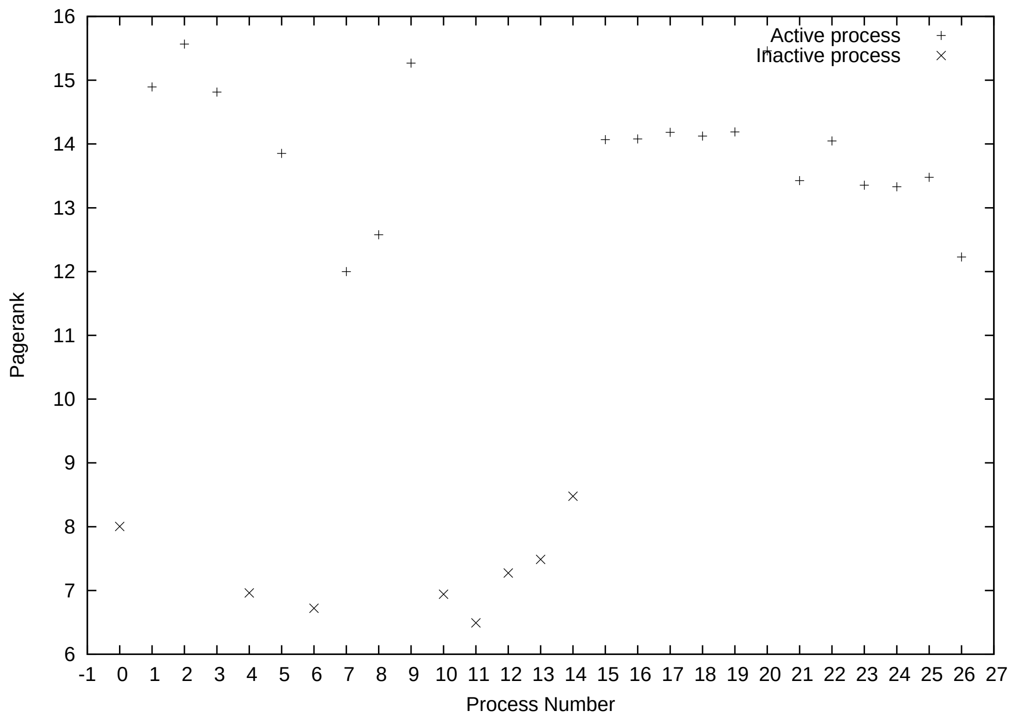

Figure 2: PageRank of process data structures in a test memory image. Using this metric the active and inactive processes are clearly separable.

Beyond the promising use of type graphs to weed out false positives from signature scans, we believe that graphs of data structure instances in memory may have other practical uses. For example, by comparing the type graphs between multiple revisions of an operating system, one could create a measure of the relative complexity of each version. It may also be possible to group structures into strongly connected clusters, and thereby infer logical groupings for functional components in the software. We hope to investigate these uses and many more in the future.

References

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[2] B. Dolan-Gavitt. PDBparse. `http://code.google.com/p/pdbparse/`.

[3] Microsoft Corporation. PDB files. `http://msdn.microsoft.com/en-us/library/yd4f8bd1(VS.71).aspx`.

[4] Microsoft Corporation. Windows research kernel. `http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.mspx`.

[5] A. Walters. The volatility framework: Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`.