

Hidden Processes Detection using the PspCidTable

Mihir Nanavati, Bhavesh Kothari
MIEL Labs
MIEL e-Security Pvt. Ltd., Mumbai

Abstract

Malware needs to be executing on a system before it can perform any damage to it. An infected file cannot damage the system unless it gets executed in some or the other manner. As a result, the detection of any malware that is running on the system is of paramount importance. The most common manner in which a binary is executed is in the form of a process. Rootkits try and hide the presence of this process from the system. In this paper, we discuss cross-view detection of hidden processes in the Windows kernel and try and outline some of the information necessary to perform it successfully. We also talk about the problems inherent to this methodology and the approach that we have taken with Helios for the detection of processes.

1. Introduction

Modern operating systems such as Windows are multi-tasking operating systems allowing a number of processes to be running simultaneously. When a binary is executed, a new process is created and it is executed in the context of that process.

The list of processes can be viewed using the Windows Task Manager. To a majority of users, the list of processes signifies all the activity that is occurring on the system and they are wary of unknown processes. Stealth rootkits that hide processes from

the user are a very real and very dangerous threat.

There are a several ways in which a process can be hidden from the user – using function hooks or by delinking the process object from a list maintained in the kernel. The latter of these is called DKOM [1].

Common evasion and detection techniques for registry keys are discussed in the paper about Strider Ghostbuster by Wang, et al. [2]. The most commonly used detection method is termed cross-view detection. In this method, an untrusted or “high-level” view of the data is obtained. This gives us the data that is visible to operating system tools such as the task manager. A second trusted view, which is termed the “low-level” view, is obtained directly from kernel structures without using the Windows API. The two views are then compared, and the discrepancies are marked as possible hidden entries. In most cases, a hidden entry will be the result of rootkit activity. It is possible that new processes being created or old ones being destroyed in the duration between the two scans are the cause of certain discrepancies. In this case, the scan can be repeated a number of times and only the discrepancies which are consistently observed are marked as malicious processes.

The trusted view of the running processes can be obtained primarily in two ways. An on-demand scan which only runs and searches through the kernel structures

when requested is one way. The other involves hooking the thread scheduler function for swapping threads, i.e. – setting a SwapContext [3] hook. This is a real time scan which flags an alert the moment a thread that does not belong to any visible process is swapped in.

In this paper we would like to focus on on-demand scanning and some of the structures that are necessary to understand for cross-view detection. We would also like to discuss a basic detection technique similar to what we have used in Helios and the test results.

2. Processes and Threads

A process is analogous to what is commonly called a program. The data about a particular process is contained in its Process Environment Block. Each process is stored in an executive object in the kernel known as the EPROCESS structure. A process may contain a number of executable threads and is basically a container for all these threads. Along with details such as the process identifier (PID) and the name, the EPROCESS structure also contains details of the threads of that process.

Threads are the units of execution in Windows [4]. This means that the thread scheduler does not care about processes and swapping in and out of processes. It is concerned with the threads and allocates a certain fraction of CPU time, known as a time slice, to each thread regardless of which process the thread belongs to. The threads are stored in kernel executive objects known as ETHREADs. Each thread has a thread execution block which contains data specific to that particular thread. The threads are swapped in and out by the thread scheduler in a way that

appears as if they are executing simultaneously.

2.1 Process Detection

A number of process detection techniques have been discussed by MS-Rem [5]. One of the first on-demand scans for hidden processes was using a brute force scan on the PIDs. The scanner will call the NtOpenProcess function sequentially with every PID from 0 to 16860. Every PID that has an associated process will fail to open. The list of all such failed calls is maintained. There should be a unique process for each of the PIDs on the list in the data obtained in the high-level view [6].

The drawback of this method is that the PID need not be unique. Hence, a process can change its PID to the PID of another existing process and evade detection. Disassembly of the function shows that it contains a reference to the PspCidTable.

The PspCidTable contains handles to all the processes and threads that are present on the system. It is used by the thread scheduler to keep track of which threads are to be scheduled. As a result, when objects are removed from it, the thread scheduler no longer remains aware of their presence and does not allocate any CPU time for that thread. This process, though officially running, cannot perform any operations now.

The FuTo [7] rootkit in normal mode removes the reference to its EPROCESS structure from the PspCidTable. As the thread scheduler allocates time slices on a thread basis, it does not need to be aware of the processes that are running. Hence, removing these entries does not impair the functioning of the process in any way.

On the other hand, when the FuTo rootkit is run with the flag `-phng`, it removes the references to both the `EPROCESS` and `ETHREAD` structures from the `PspCidTable`. As the thread scheduler does not schedule these threads any longer, the process remains running, but no operations can be performed on it.

The kernel maintains link lists of process and thread data. Each `EPROCESS` structure is part of the process list, while the `ETHREAD` structures are part of several thread lists. The detection engine can walk the entire list of threads and processes looking for some that may have been removed from the `PspCidTable`, but are still being executed on the system.

2.2 Structure of the PspCidTable

The `PspCidTable` has a structure similar to that of a handle table. In Windows XP, it may have 1-3 dimensions, depending on the number of processes that are currently being executed on the system. It starts off by being a single dimension table with 512 entries for the executive objects. However, as the number of processes and their threads increases, 512 entries may be too few to contain all their references. The table now automatically becomes a 2 dimension table, where the first 512 entries are pointers to other tables which contain the actual links to the executive objects.

In case of a 3 dimensional `PspCidTable`, the first 512 entries are pointers to the second level tables. Each of the 512 second level tables, in turn, contains 512 entries pointing to the third level table. The 512^2 third level entries are each a table of 512 pointers to the actual executive objects.

2.3 Kernel Link-Lists

The `EPROCESS` structures are part of the `ActiveProcessList` and `SessionProcessList`, while the `ETHREAD` structures are part of the `ThreadList`, the `ThreadWaitList` and the `ThreadQueueList`.

The link lists are circular, doubly-linked lists. The link to the next element is called the forward link or `Flink` and the link to the previous element is called the backward link or `Blink`.

Each `Flink` or `Blink` points not to the next `EPROCESS` or `ETHREAD` structure in the list, but to the `Flink` or the `Blink` of the next `EPROCESS` or `ETHREAD` structure.

As it is a circular link list there is no head or tail and it can be traversed equally effectively by starting off at any random `EPROCESS` or `ETHREAD` structure and proceeding in either direction.

2.4 Building a Process List

The rootkit detection engine starts off by reading in the flag in the `PspCidTable` signifying the number of dimensions it is. For the sake of illustration, let us assume a 2 dimensional table. Every non-null entry in the first table points to another table of entries. Each of the non-null entries in this table points to an `EPROCESS` or `ETHREAD` structure. Thus, the initial list of the processes and threads can be obtained from the `PspCidTable` itself. Each structure is determined to be an `EPROCESS` or an `ETHREAD` on the basis of the tag in the header [8].

Once the list is obtained, one needs to search for processes that may have been removed from the table. This can be done by walking through the `ActiveProcess` and

SessionProcess link lists that are present in every EPROCESS structure.

It is also necessary to verify that all the threads that are being scheduled have a valid parent process. As a result, for each ETHREAD structure, the parent EPROCESS structure is obtained. If this results in the finding of a process that is not in the list of processes, it is likely that the process has been hidden as a result of rootkit activity.

The ETHREAD structures also members of different lists depending on the state of the thread. Walking of these link lists may yield ETHREAD structures that have been removed from the PspCidTable, and their parent EPROCESS structures.

A comprehensive list of processes running on the systems is now available for comparison.

3. Locating the PspCidTable

The location of the PspCidTable can be obtained from the Processor Control Block [9]. The KPCR_BLOCK is located at the memory location `0xffdff000`. It contains a member known as KdVersionBlock at an offset of `0x34`. This is a pointer to the `_DBGKD_GET_VERSION` structure. There are both 32 and 64-bit versions of this structure, however, the 32-bit version has been deprecated and the 64-bit version is in use.

This structure is `0x28` bytes and is usually immediately followed by the `KDDEBUGGER_DATA64` structure. The `KDDEBUGGER_DATA64` has a tag in its first four bytes that is matched with 'KDBG'.

In case the tag is not matched it is possible that the `KDDEBUGGER_DATA64` structure is not immediately after the `_DBGKD_GET_VERSION64` structure.

The last member (`0x20` offset) of `_DBGKD_GET_VERSION64` is a pointer to the list which has only one element - the `KDDEBUGGER_DATA64` structure. Both the Flink and the Blink of that list point to this structure. By reading these values we get a pointer to the memory location of the actual `KDDEBUGGER_DATA64` structure.

This structure has a pointer to the `PspCidTable` at the offset of `0x58`.

4. Cross-View Detection in Helios

Helios performs an online scan for hidden processes on the system. The "low-level" data is obtained by reading kernel structures such as the `PspCidTable` and by walking the various link lists present in the executive structures such as `EPROCESS`, and `ETHREAD` present in it. While these structures could be read directly using a device driver, Helios uses the `ZwSystemDebugControl` to read the virtual memory in a manner similar to that used by `WinDbg`.

The names of the processes and the process IDs are then obtained from the `EPROCESS` structures. Some rootkits may blank these values out, making them more difficult to identify.

The "high-level" view is obtained using Windows API functions such as `ZwQuerySystemInformation` and then parsing the list of processes returned in the buffer. This is similar to the method used by Task Manager in displaying a list of processes to the user.

This technique is successful in detecting most of the current rootkits that hide malicious processes such as FuTo, rkDemo [10] and phide_ex [11].

5. Limitations

There are so many different ways in which processes can be concealed that finding a single fool-proof method that detects every rootkit that hides processes is well nigh impossible. Since removing entries from the PspCidTable render the process and threads harmless, it is a good place to search for rootkits. However, it is likely that rootkits will evolve to re-implement the thread scheduler so that they do not need to rely on the Windows thread scheduler to obtain CPU time as used to bypass Klister [12] in Windows 2000 in the phide2 engine [13]. The thread entries can now safely be removed from the PspCidTable and still be executed bypassing a large number of rootkit detection engines.

As evasive techniques evolve and continue to get more and more sophisticated, any rootkit detector will also have to evolve to counter these threats.

6. Conclusion

In this paper we discussed how to locate rootkits that are hiding processes on a system using the kernel executive structures and the PspCidTable. There are a number of other ways that can also be used for the same effect, and there is no single best methodology. Rather, a combination of methods which access data from a number of locations raise the bar for evasion. To successfully evade detection, the rootkit would need to remove itself from all the locations while still remaining active and listed for

scheduling – a feat few rootkits today can achieve.

References

- [1] G. Hoglund and J. Butler. Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional, August 2005.
- [2] Y.M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. Microsoft Research Technical Report MSR-TR-2005-25, February 2005.
- [3] J. Butler, J. L. Undercoffer, and J. Pinkston. Hidden processes: The implication for intrusion detection. In Proceedings of the 2003 IEEE Workshop on Information Assurance.
- [4] M. Russinovich and D. Solomon. Microsoft Windows Internals (4th Edition): Microsoft Windows Server 2003, Windows XP, and Windows 2000. Microsoft Press, January 2005.
- [5] Ms-Rem. Detection of Hidden Processes. <http://ms-rem.dot-link.net/hiddndt/hiddndt.htm>
- [6] P. Silberman. FuTo. Uninformed Journal, Volume 3 Article 7, Jan 2006
- [7] J. Butler and P. Silberman. Fu and FuTo. https://www.rootkit.com/vault/fuzen_op/FU_Rootkit.zip
http://www.rootkit.com/vault/petersilberman/FUTo_enhanced.zip
- [8] S. Schreiber. Undocumented Windows 2000 Secrets: A Programmer's Cookbook. Addison-Wesley Professional, May 2001.
- [9] A. Ionescu. Getting Kernel Variables from KdVersionBlock, Part 2. <http://www.rootkit.com/newsread.php?newsid=153>
- [10] MP_ART. RkDemo.
- [11] PE386. phide_ex.
- [12] J. Rutkowska. Klister <http://www.rootkit.com/project.php?id=14>
- [13] 90210. Bypassing klister 0.4 with no hooks or running a controlled thread scheduler. <http://hi-tech.nsys.by/33/>