Evasive Malware Detection Using Groups of Processes

Gheorghe Hăjmăşan[1,2](✉), Alexandra Mondoc[1,3], Radu Portase[1,2], and Octavian Creţ[2]

[1] Bitdefender, Cluj-Napoca, Romania
{amondoc,rportase}@bitdefender.com
[2] Technical University of Cluj-Napoca, Cluj-Napoca, Romania
{Gheorghe.Hajmasan,Octavian.Cret}@cs.utcluj.ro
[3] Babeş-Bolyai University, Cluj-Napoca, Romania

Abstract. Fueled by a recent boost in revenue, cybercriminals are developing increasingly sophisticated and advanced malicious applications. This new generation of malware is able to avoid most of the existing detection methods. Even behavioral detection solutions are no longer immune to evasion, mostly because existing solutions focus on the actions or characteristics of a single process. We propose shifting the focus from malware as a single component to a more accurate perspective of malware as multi-component systems. We propose a dynamic behavioral detection solution that identifies groups of related processes, analyzes the actions performed by processes in these groups using behavioral heuristics and evaluates their behavior such that even evasive, multiprocess malware can be detected. Using the information provided by groups of processes, once a malware has been detected, a more comprehensive system cleanup can be performed, to ensure that all traces of an attack have been removed and the system is no longer at risk.

1 Introduction

Malicious software has become the foundation of a highly profitable industry. To maximize profit, malware authors are developing increasingly sophisticated attacks. The new breed of malware is able to avoid static detection through various methods, like obfuscation or encryption. To make detection even more difficult, thousands of new malware or variants of existing malware are being released every day. Consequently, dynamic detection has become more important, representing a last line of defense in security solutions.

Currently, the majority of dynamic malware detection techniques evaluate the behavior of a process and, using a set of rules, decide if that process is malicious or not. The rule set must accurately differentiate between malicious and non-malicious processes. Because a balance between detection rate and number of false positives must be assured, a dynamic detection system can not be too aggressive when evaluating a single process. Advanced malware may take

S. De Capitani di Vimercati and F. Martinelli (Eds.): SEC 2017, IFIP AICT 502, pp. 32–45, 2017.
DOI: 10.1007/978-3-319-58469-0_3

advantage of this lack of aggression. They can evade being detected by separating malicious actions into multiple processes through process creation or code injection. This separation causes current dynamic detection systems to be unable to detect some of the malware components or, even worse, not to detect the malware at all. This is a major issue, because if a malware attack is only partially detected and the malicious components are not entirely removed from a system, they will continue to represent a serious security risk for the user.

We propose a behavioral detection solution that overcomes the issue of detecting evasive malware. We propose renouncing the current view of malware as single component systems and adopting a more accurate and comprehensive, multi-component based, method of evaluation and detection.

The following sections present a method to detect malicious groups of processes instead of single malicious processes. This research will provide a method for constructing such groups, together with a way to evaluate their actions so that malware groups can be detected. We also present a way to clean the infected system based on the actions performed by the processes in the detected group.

This paper is organized as follows: Section 2 presents the current state of research concerning behavioral malware detection and how most common solutions can be evaded. The proposed solution is described in Sect. 3 and the results of the proposed solution are presented in Sect. 4. The conclusions are mentioned in Sect. 5.

2 Related Work

An approach used in behavioral malware detection consists of extracting features based on the API calls performed by an analyzed sample. Devesa et al. [2] propose identifying which actions were performed, based on API calls records. These actions represent features, used to classify a sample as malicious or clean.

Constructing graphs based on the relations between system calls represents another approach in behavioral malware detection. Elhadi et al. [3] propose creating data dependent graphs, with nodes representing system calls and the edges, relations between their parameters or return values. An algorithm based on the Longest Common Subsequence is used to match the obtained graph to those of known malware stored in a database. Behavior graphs are also used in [7]. Compared to other similar solutions, the solution proposed by Kolbitsch et al. has the advantage of matching the behavior graphs in real time, providing protection on the end host. Naval et al. [10] propose representing the behavior of a sample as an ordered system call graph and extracting relevant paths, which are considered features used for classifying the sample as malicious or benign.

Most dynamic malware detection solutions that focus on analyzing the behavior of individual processes are highly vulnerable to a certain type of evasion that is increasingly used by sophisticated malware and advanced threats. The evasion mechanism is quite simple: instead of executing all the malicious actions from a single process - which could be more easily detected by advanced security solutions - the malicious payload is distributed to multiple, distinct processes, and may be executed over a long period of time. Because behavior based detection solutions can not usually detect a process based on a single action, multiple individual processes, each performing a smaller set of actions, may go unnoticed, allowing the malware to achieve its target goal undetected.

Ma et al. [8] developed a prototype tool, working at compiler level, that can generate multiple "shadow" processes from the original malware code. Each "shadow" process executes some of the payload, such that the original behavior of a process remains unchanged. Various methods to deliver malware distributed into multiple files are presented in [11]. Another method of distributing the malicious payload, presented in [4] consists of injecting parts of the payload into clean processes running on a system. This approach makes cleanup more difficult because, if only one injected process is terminated, the malware is capable of reinstantiating itself from another injected process. The distinct malicious processes may communicate using traditional inter process communication, supported by the operating system, or through purposely implemented special mechanisms.

This evasion mechanism is extremely effective especially against detectors based on API or code flow graph. Since the API calls are distributed to multiple distinct processes, this type of detectors may have difficulties in matching the obtained graphs, or may be unable to do so. The effectiveness of distributing malicious behavior to multiple processes is also recognized by [6,12].

A solution designed to combat multi-process malware is proposed in [5]. In the approach presented by Ji et al., the actions performed by each process are represented as feature vectors and then correlated with the actions performed by its child processes. The correlation phase in malware detection may be a complex problem, both in terms of implementation and efficiency. Additionally, this solution does not consider code injection when correlating processes.

Evasion mechanisms such as those previously described represent a strong argument to show that behavioral-based security solutions need to evolve past analyzing a single process, individually and in isolation from other entities. Focus should shift to developing more advanced security solutions, capable of analyzing each process in the broader context of all the processes executed on a computing system and taking into account any relations between them.

3 Proposed Solution

A high level view of the proposed solution, illustrating its major components and the interactions between them is presented in Fig. 1. Our implementation is intended for the Windows Operating System (OS), but the proposed approach may be applied for other operating systems.

The essential requirement for a behavioral detection solution is to monitor the actions performed by processes. This is implemented within the *Event Interceptors.* They use mechanisms specific to the Windows OS and are located both in Kernel Mode and User Mode (UM). In Kernel Mode, the solution uses a

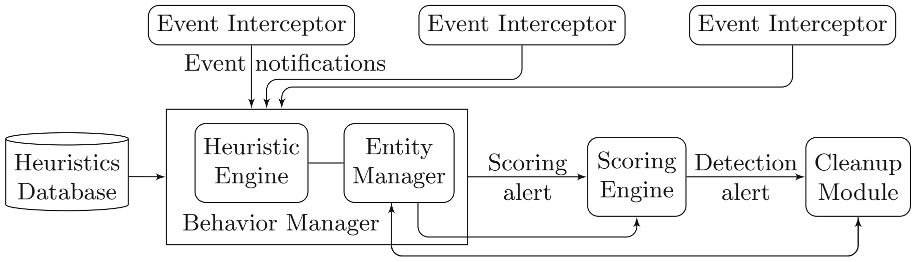

Fig. 1. Behavioral detection solution

minifilter driver [9] that registers callback routines, which are notified whenever changes occur in the file system, registry keys or when processes are created. At User Mode level, the actions are filtered using API interception (hooking) through a DLL injection [1] into the monitored process. The intercepted actions are encapsulated in events and sent to the *Behavior Manager*, consisting of the *Heuristic Engine* and the *Entity Manager*.

The detection is based on behavioral heuristics located in the *Heuristic Engine*. A heuristic is an algorithm that analyzes the actions performed by processes, using the intercepted events. Some heuristics are defined in signature files and are retrieved by the engine from the *Heuristics Database*.

The *Entity Manager* uses information provided by the Event Interceptors, together with information from some heuristics (e.g. for detecting code injection) to manage the processes and groups on a system and their relations.

When a heuristic decides a malicious action has been performed, it sends an alert to the *Scoring Engine*, where it is evaluated. This component computes scores for the entities that caused the alert and decides whether they are potentially malicious. If a process or group of processes is considered malicious a *detection alert* will be sent to the *Cleanup Module*. This module is responsible with taking anti-malware actions against the target entity. The *Cleanup Module* and the *Scoring Engine* use the information provided by the Entity Manager in order to identify all the relations between the malicious entities.

In a broader perspective such a solution should be integrated (as a last line of defense) in a modern security application, together with other components such as URL blocking, firewall, classic AV signatures, etc.

3.1 The Management of Groups

In order to function effectively, the solution must have a complete overview of the running processes. To accomplish that, the *Entity Manager* maintains a collection of processes executing on the client system. The Entity Manager dynamically updates this collection to reflect the addition of new processes in response to process creation, and the removal of other processes in response to process termination. The Entity Manager divides the processes in the collection into one or multiple *groups* and maintains a set of associations indicating the

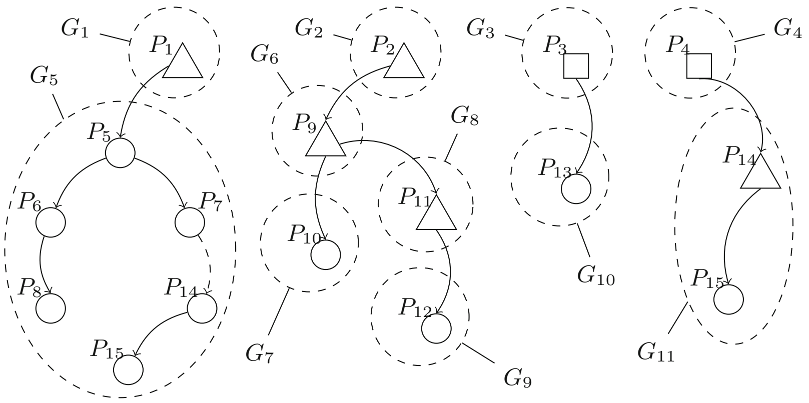

Fig. 2. Groups of processes

groups each of those process belongs to. An example illustrating multiple groups of processes is presented in Fig. 2.

Categories of Processes. Processes are divided into three distinct categories: *group creators* - illustrated using triangles, *group inheritors* - circles and *unmonitored processes* - squares. By assigning a category - or a role - to each process, the groups of processes are much easier to identify and manage. Smaller groups, consisting of processes that are actually related, can be created, avoiding the creation of a single, large group per system. The category which a process belongs to is identified based on certain features of the respective process. Examples of such features are the file path, the digital signature or a hash computed for the executable file corresponding to the process.

Relations of Processes. The solid arrows indicate process creation, while the dashed arrows indicate code injection. The direction of each arrow indicates the direction of the relationship between the respective entities. For example, process P_6 is a child of process P_5 while process P_7 has injected code into process P_{14}. Groups of related processes are represented as dashed lines, encircling those processes, and are denoted as G_i, $i \in \{1, 11\}$. For example, P_1 is the sole member of group G_1, while G_5 contains processes $P_5 \ldots P_8$, P_{14} and P_{15}.

Group creators are processes that are known to create other processes, not necessarily related to them. As their name suggests, whenever a process from this category spawns a process, a new group will be created, initially consisting of the child process. This category includes, among others, *winlogon.exe*, *svchost.exe*, *cmd.exe* and other processes or services of the OS, Windows Explorer, Total Commander and similar file manager applications, Internet Explorer, Firefox, Chrome and other browsers. When a group creator spawns a process a new group is created (e.g. group creator P_1 creates group G_5 when it spawns P_5).

Processes that are *unmonitored* by the security application include the various components of the security solution and certain components of the OS, for example *csrss.exe* and *smss.exe* on the Windows OS. These processes are implicitly treated as group creators.

The *group inheritor* category includes the majority of user processes, as well as processes that are unknown or are not identified as group creators. Whenever a group inheritor spawns a process or injects code into another process, the other process is included in the same group as the group inheritor (e.g. process P_6 is included in the same group with its parent process, P_5; P_{14} is included in the same group as process P_7, as a result of receiving injected code from P_7).

The category of a process is updated in response to certain events or when it becomes part of a group. In Fig. 2, process P_{14} was initially a group creator, as shown in group G_{11}. At a later moment, it received code injected by process P_7, a member of G_5. As a result, process P_{14} was included in the group of the injector process, G_5, and was re-marked as a group inheritor.

A process may also simultaneously belong to multiple groups, due to code injection. However, such situations are not so frequent. In the example illustrated in Fig. 2, process P_{14} has become a group inheritor and is included in both G_5 and G_{11} groups, as described above. When process P_{14} - now a group inheritor - spawns the new process P_{15}, the latter will be included in both the G_5 and G_{11} groups. In other words, changing the category of a process impacts how the processes it spawns or injects code into are handled.

The groups of processes are managed by the Entity Manager, which receives notifications from various Event Interceptors whenever an event related to the life cycle of a process occurs. Process life cycle events consist of process creation, code injection and process termination. If the event indicates the creation of a new process, the Entity Manager determines whether the parent process is a group inheritor or not, in order to assign the newly created process to the appropriate group. If the parent is a group inheritor, the manager will add the child process to the parent's group and will mark it as a group inheritor. Otherwise, the manager determines if the parent process is a group creator. If so, a new group will be created and the child process will be added to that group.

Figure 3 presents a real-world example using a TrojanSpy.MSIL[1] malware. During the two minutes the sample was run, it launched multiple processes, including *cmd.exe* and *reg.exe* (used to modify registry). Under normal circumstances, *cmd.exe* is a group creator, but because the first process in the group is a group inheritor, all its descendant processes become group inheritors.

If the process life cycle event is a code injection, the Entity Manager will determine if it represents a trusted injection. Usually, each code injection event is considered suspicious, possibly indicating a malicious action. However, some processes of the OS may, in some specific situations, legitimately inject code into other processes. These situations should not be considered malicious in order to avoid false positives. The Entity Manager attempts to match the details of the code injection event to a *whitelist*, containing details of legitimate injections.

[1] MD5 hash: 0x143FCC07CEB0F779FF1E204CEF4A20D6.

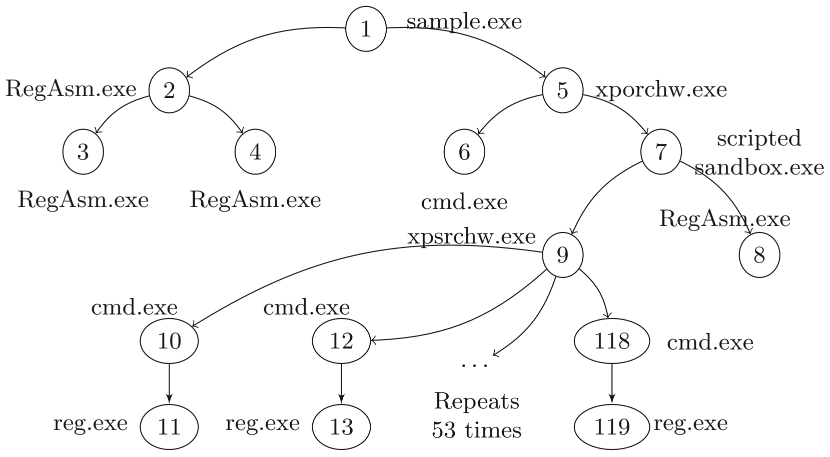

Fig. 3. TrojanSpy:MSIL malware

If the current event is not recognized as a known kind of legitimate injection, the Entity Manager will add the processes receiving the injected code to the group of the process performing the code injection. Then the injected process is marked as a group inheritor, even if initially it was categorized as a group creator.

If the analyzed event indicates the termination of a process, that process is marked as dead. However, it will not be removed from a group until all the other processes in that group have terminated. This strategy will allow a security solution to perform a comprehensive cleanup of the protected system, eliminating even evasive malware that, for instance, only spawn child entities and then exit.

3.2 Heuristics

The proposed security solution relies on *behavioral heuristics* to analyze the actions performed by processes, based on the information provided by Event Interceptors. Whenever a heuristic identifies that a targeted action is being performed, it triggers an alert to the Scoring Engine. Each alert consists of several information about the detected action and the entity that performed it. An alert also has an associated score, that is used to evaluate the potential of a process or group of being malicious.

Some of the actions that can be identified using heuristics are: creating a copy of the original file, hiding a file, injecting code into another process, creating a startup registry key such that the malicious application will be executed after a system restart, deactivating some critical OS functionalities (e.g. Windows Update), terminating critical processes or processes associated with security solutions or modifying an executable file belonging to the OS.

Figure 4A illustrates a heuristic that listens for events to identify six actions in a certain time order. If these actions are identified, the heuristic will trigger

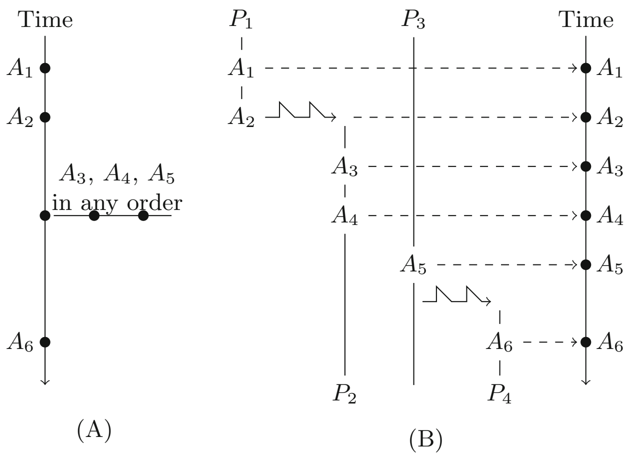

Fig. 4. Heuristic's logic example

an alert. In the proposed solution the logic of the heuristic is implemented in two ways, as function-callbacks or as heuristic signatures, depending on the complexity of the heuristic. In the first case the heuristics are procedures (functions) that are called whenever an event that they registered for occurs. The second one uses signatures to store the logic of simpler heuristics and an engine that tries to match the signatures with the intercepted events.

If a heuristic listens for actions performed only by a process it is called *process heuristic*. If it listens for actions performed by all the processes inside a group it is called *group heuristics*. An example of group heuristic is illustrated in Fig. 4B. Whenever processes $P_1 \dots P_4$ perform actions $A_1 \dots A_6$ in a specific order, such a heuristic will trigger an alert for the group that contains, among others, processes $P_1 \dots P_4$. Process creation is illustrated as a zigzagged arrow. The life history of each process is represented as a solid vertical line. For example, process P_1 terminates after it spawns process P_2. Process P_3 becomes a part of the illustrated group in response to receiving injected code from P_2. Some actions of the respective processes are not part of the heuristic presented in Fig. 4B, for example the spawning of process P_4 by process P_3, mainly because they are not invariants between multiple executions.

The sequence of actions $A_1 \dots A_6$ describes a ransomware attack. Ransomware is a type of malware that encrypts a set of files on the user's computer and demands a ransom payment in order to recover the files. It has become very popular recently among malware authors, because it represents an almost sure source of revenue. In this example, the malicious actions are distributed among a group of processes $P_1 \dots P_4$. Each member of the malicious group performs only a small amount of these actions. The actions performed by the ransomware are: A_1: dropping a copy of itself on disk, A_2: launching a copy of itself, A_3: deleting backup (shadow) files, A_4: injecting code into another process, A_5: enumerating and encrypting files, A_6: displaying a message demanding the ransom.

Individually, each action $A_1 \ldots A_6$ may be performed legitimately by a clean application. For example, dropping a copy of itself on disk or launching it (actions A_1 and A_2) are commonly performed by installers. Additionally, deleting backup files (action A_3) may be performed by certain tools or the Operating System to free disk space. Many clean applications perform code injection (action A_4) for various purposes, such as adding functionalities to an existing, previously released product. Most applications for management of media libraries can legitimately enumerate or modify certain files (action A_5). Finally, displaying a message to the user (action A_6) is specific to almost every GUI application.

An experienced behavior-based detection researcher may observe that a more generic heuristic is possible, that triggers when the group executed the action A_3 or A_4, but the presented heuristic was extended for the sake of the example. Also, one may observe that the flexibility granted by using such heuristics may allow detecting various versions, variants or an entire class of malware. For example, the heuristic presented in Fig. 4 triggers an alert for the CTB Locker[2] sample, whose group is presented in Fig. 5. Regardless of how the actions $A_1 \ldots A_6$ are distributed among processes within the group, if they are executed in the same order as presented in Fig. 4A, the heuristic will trigger an alert on the group.

Heuristic's Evaluation. The Scoring Engine receives *scoring alerts* from the Heuristic Engine whenever a heuristic determines that the occurrence of an event indicates a malicious action. Based on these alerts, the Scoring Engine maintains and updates the *aggregated scores* for the involved entities, process or group. Depending on the heuristic, the alert can influence the aggregated scores of a single process, of a group of processes or of both types of entities.

Using these scores, the Entity Manager determines whether a malware is present on the client system (e.g. a score threshold is reached). When this happens a *detection alert* is sent to the Cleanup Module, that will take the actions necessary to remove the malicious component from the system. Using

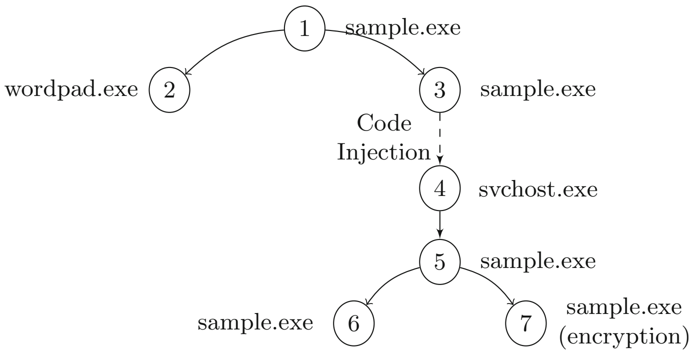

Fig. 5. CTB Locker ransomware

[2] MD5 hash: 0x82F941FBD483E0684DAED99F006488F1.

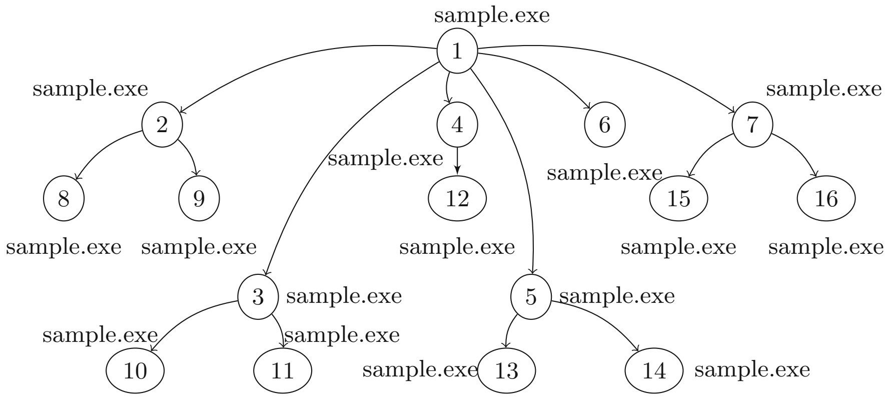

Fig. 6. Trojan-PSW malware

these evaluation methods, even if malicious actions are distributed between several members of a group, and the aggregated scores corresponding to each individual processes are not sufficient to trigger a detection, the group-wide score may exceed the detection threshold. This is very useful for malware such as the Trojan-PSW[3] sample, illustrated in Fig. 6. This malware creates many processes from the same executable file, each having different command line arguments, distributing its payload in this way.

3.3 Remediation

In order to assure the best protection of a system, once a malicious entity is detected, whether it is a process or group of processes, all traces of that entity must be removed from the system and any changes performed by it must be undone. The *Cleanup Module* is responsible for taking such actions, based on information received from the Scoring Engine and the Entity Manager.

When the module receives the detection alert, it will first identify the process that triggered the detection and determine if it belongs to a single group or to multiple groups. If the suspect process belongs to a single group, the module will proceed to clean the entire group of that process, by applying the appropriate cleanup operation on each member of that group. Cleanup operations usually start with suspending or terminating the execution of the targeted entity. Then, the operation may continue with deleting the disk files that contain the code of that entity and undoing or rolling back a set of changes performed by the respective entity, such as changes to a registry of the OS or to the file system. In some situations, malicious activities may be related to a code injection event. In that case, the Cleanup Module terminates the process that received the injection. Special attention should be given to situations where a malware uses a clean process of the OS to carry out part of a malicious attack through code injection.

[3] MD5 hash: 0x609614B508622E90EEEDAA875226FEA4.

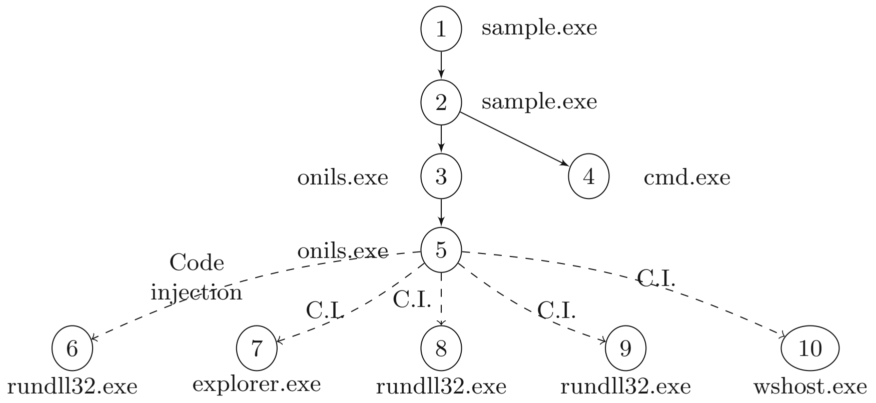

Fig. 7. ZBot malware

In this case, the module may terminate the respective clean process, but it should not delete its executable file so that no damages are made to the OS. An example for this case is the ZBot[4] malware, illustrated in Fig. 7, which injects code into multiple clean processes.

If the suspect process belongs to multiple groups, the Cleanup Module attempts to identify which of those groups is malicious. For example, it could determine how the suspect process became a member of each group: by process creation or code injection. Next, by identifying which heuristic triggered the detection, the Cleanup Module could determine what action the suspect process has performed. For example, we consider a suspect process that is member of a first group via process creation and a member of a second group via code injection. The Cleanup module will attempt to determine the source of the code that was executing when the scoring alert that caused the detection was triggered. If the alert was triggered while the suspected process was executing code from its main executable module, the Cleanup Module will determine that the first group is malicious. Otherwise, if the injected code was being executed, the Cleanup Module will determine that the second group is the malicious one. If the malicious group is successfully identified, the module will proceed with cleaning that group. Otherwise, it will only clean the suspect process, to prevent potential data loss for the user in case of a false positive detection.

4 Technical Results

When evaluating a security solution the detection rate, false positive rate and performance impact are the most important criteria to be considered. A good security solution must have a high detection rate, a low false positive rate and unnoticeable performance impact.

[4] MD5 hash: 0x43A6DD7D5BE93F4E5224940C67E40FF8.

4.1 Detection Tests

A comparison between the detection rate of the group based approach and a non group based solution is presented. The detection tests were performed in a virtual environment consisting of machines running Windows 8.1. Each sample was run for two minutes in the virtual machine, then the results were collected and the execution was ended. For false positives tests, each sample was run for ten minutes in the virtual machine before terminating the execution.

For the detection test, two malware collections were used, the first consisting of ransomware that were collected in November 2016, while the other contains malware samples collected from various sources like: honeypots, spam email attachments, infected WEB sites and URLs used to spread malware in November 2016. The clean samples (for the false positive test) are popular applications used in 2016.

Table 1. Malware detection test

Samples	Detected (no groups)	Detected (no groups)	Detected (with groups)	Detected (with groups)
47933	37054	77.3%	42142	87.91%
16490	13084	79.34%	13935	84.5%

Table 1 shows that the detection was improved for both collections with 10.61% and 5.16%. This shows that at least 5% of the malware in both collections are multi-component or multi-process, thus proving the need of changing the detection approach to a group based solution. This amount may not seem much at first glance, but such small differences make the distinction between an average security solutions and a good, competitive one.

Table 2. False positive test

Samples	Detected (no groups)	Detected (no groups)	Detected (with groups)	Detected (with groups)
1128	10	0.88%	10	0.88%

The results of the false positives test, presented in Table 2, show that the number of false positives does not change when augmenting the security solution with group awareness. This is due to the fact that the groups generated for legitimate applications usually contained a small number of processes with few triggered heuristics.

4.2 Limitations of the Solution

The implementation of the solution involves maintaining in memory a set of information associated to each process in a group until the group is terminated. For some samples, such as the TrojanSpy.MSIL sample the memory requirements are high. This can be prevented by detecting the sample before the process group contains too many processes or for clean processes, by simply making that process a group creator.

Clean processes are added to malware groups because malware use such processes to perform different actions (e.g. *reg.exe* to access the registry). This problem indicates that a *whitelist* is needed, that will be consulted when cleaning the infected system to prevent any data loss for the user or producing any damages to the Operating System.

The solution can only detect samples which interact on the current machine. If by some means a process uses an external (i.e. not on the same machine) communication channel to force the creation of another process on the original machine the Entity Manager can not link the parent with the child and it is not able to create the group correctly.

Finally the solution is limited by the platform it runs on. Because Windows does not keep a strict relation between child processes and parent processes, managing groups can prove to be difficult, requiring OS specific knowledge. Furthermore, because Windows allows code to be injected in a trivial way and does not provide a synchronous notification for when injections occur, detecting all code injection methods is also considerably hard. The proposed solution attempts to solve this issue by identifying the most common methods for injecting code through dedicated heuristics.

5 Conclusions

We highlighted the problem of evasive, multi-process malware and proposed shifting the focus from evaluating the behavior of individual processes to evaluating and correlating the actions of related processes. We presented real-world malware samples, in order to better exemplify the behavior of multi-process malware. The proposed solution detected all these samples and constructed their groups correctly.

We described how groups of related processes are constructed, by dividing the processes into *creators* and *inheritors*. We presented the way groups are influenced by process creation and code injection events. We introduced group-based behavioral heuristics, described how the behavior of processes and groups is evaluated and how detected entities can be cleaned.

A major contribution of our solution is that it automatically correlates the behavior of individual processes within a group, thus eliminating the need for a distinct correlation phase, as presented in [5], which is both costly and complex. As a result, the heuristics are easier to develop, the evaluation is more straightforward and cleanup is better performed.

We implemented the presented concepts into a behavior-based solution and compared this approach to a non-group solution. The improvement was quite consistent: the detection rate was increased with over 10% for the ransomware samples test, a type of malware known to be highly evasive (multi-process).

References

1. Blunden, B.: The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System. Jones and Bartlett Publishers Inc., USA (2009)
2. Devesa, J., Santos, I., Cantero, X., Penya, Y.K., Bringas, P.G.: Automatic behaviour-based analysis and classification system for malware detection. In: ICEIS 2010 - Proceedings of the 12th International Conference on Enterprise Information Systems, AIDSS, Funchal, Madeira, Portugal, 8–12 June 2010, vol. 2, pp. 395–399 (2010)
3. Elhadi, A.A.E., Maarof, M.A., Barry, B.I.: Improving the detection of malware behaviour using simplified data dependent API call graph. Int. J. Secur. Appl. **7**(5), 29–42 (2013)
4. Ispoglou, K.K., Payer, M.: malWASH: washing malware to evade dynamic analysis. In: Proceedings of the 10th USENIX Conference on Offensive Technologies, WOOT 2016, pp. 106–117. USENIX Association, Berkeley (2016)
5. Ji, Y., He, Y., Jiang, X., Cao, J., Li, Q.: Combating the evasion mechanisms of social bots. Comput. Secur. **58**(C), 230–249 (2016)
6. Ji, Y., He, Y., Zhu, D., Li, Q., Guo, D.: A mulitiprocess mechanism of evading behavior-based bot detection approaches. In: Huang, X., Zhou, J. (eds.) ISPEC 2014. LNCS, vol. 8434, pp. 75–89. Springer, Cham (2014). doi:10.1007/978-3-319-06320-1_7
7. Kolbitsch, C., Comparetti, P.M., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: Effective and efficient malware detection at the end host. In: Proceedings of the 18th Conference on USENIX Security Symposium, SSYM 2009, pp. 351–366. USENIX Association, Berkeley (2009)
8. Ma, W., Duan, P., Liu, S., Gu, G., Liu, J.C.: Shadow attacks: automatically evading system-call-behavior based malware detection. J. Comput. Virol. **8**(1–2), 1–13 (2012)
9. MSDN: file system minifilter drivers. http://msdn.microsoft.com/en-us/library/windows/hardware/ff540402%28v=vs.85%29.aspx
10. Naval, S., Laxmi, V., Rajarajan, M., Gaur, M.S., Conti, M.: Employing program semantics for malware detection. IEEE Trans. Inf. Forensics Secur. **10**(12), 2591–2604 (2015)
11. Ramilli, M., Bishop, M.: Multi-stage delivery of malware. In: 5th International Conference on Malicious and Unwanted Software (MALWARE), pp. 91–97, October 2010
12. Ramilli, M., Bishop, M., Sun, S.: Multiprocess malware. In: Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software, MALWARE 2011, pp. 8–13. IEEE Computer Society, Washington, DC (2011)