Behavioral analysis of malicious code through network traffic and system call monitoring

André R. A. Grégio[a,b], Dario S. Fernandes Filho[a,b], Vitor M. Afonso[a,b], Rafael D. C. Santos[c], Mario Jino[b], Paulo L. de Geus[b]
[a]Information Technology Research Center (CTI/MCT), Campinas, SP, Brazil;
[b]University of Campinas (UNICAMP), Campinas, SP, Brazil;
[c]National Institute for Space Research (INPE), São José dos Campos, SP, Brazil

ABSTRACT

Malicious code (malware) that spreads through the Internet—such as viruses, worms and trojans—is a major threat to information security nowadays and a profitable business for criminals. There are several approaches to analyze malware by monitoring its actions while it is running in a controlled environment, which helps to identify malicious behaviors. In this article we propose a tool to analyze malware behavior in a non-intrusive and effective way, extending the analysis possibilities to cover malware samples that bypass current approaches and also fixes some issues with these approaches.

Keywords: Malicious software, malware behavior, dynamic analysis, network security.

1. INTRODUCTION

Malicious code spreading through Internet is currently a major threat to information systems security. Malicious code, or malware, are a class of applications that can also be called individually as *virus*, *worms*, *keyloggers*, *trojans*, *backdoors* and other kinds of programs intended to compromise systems. The increasing interaction of several computing devices with the Internet, the vast provision of services and the lack of knowledge of regular users contribute to the existing scenario of attacks through malware. The motivation behind these attacks is the economy established based on the rental of compromised infrastructures and sale of sensitive information, such as bank account numbers, passwords and credit card numbers [13] [10].

Though from time to time some malware instances come into action—sometimes causing serious incidents with global proportions, as the recent results of the Zeus [5] and Conficker [17] malware—the activities executed in the victim system can always be mapped to specific actions, such as disabling protection mechanisms (firewall, antivirus), modification of system binaries and libraries, change of registry values, opening of network ports etc.

Thus, one approach that can be used to identify malware is through the identification of its actions in the victim system through an analysis of the malicious code. Malware analysis can be done in two ways: statically and dynamically. In a static analysis, it is possible to obtain information about the malware code without executing it, whereas in the dynamic analysis the behavior of the malware is monitored during its execution. Malware analysis can help the automation of the analysis process made by antivirus enterprises and the provision of information that can be useful to create new ways of malware identification and mitigation.

Due to the problem of malware obfuscation, which leads to different representations of a program that produces the same result, the static analysis became extremely difficult and often with unreliable results. This obfuscation technique is, in part, ineffective against a dynamic analysis. There are some systems available in the Internet that perform dynamic analysis of malware, each with its technical peculiarities. Among these systems, we can cite Anubis [3], CWSandBox [24], BitBlaze [23], Ether [8] and JoeBox [14], that run the malware in a Windows XP system and monitor all the activities for a limited amount of time, generating an analysis report at the end.

All analysis approaches have shortcomings, the most relevant of them is the fact that they cannot analyze binaries that have specific detection mechanisms. These mechanisms allow the malware to realize that it is being monitored and to stop the execution or to change its behavior. In this paper we present BehEMOT (*Behavior Evaluation from Malware Observation Tool*), a system that can automatically analyze malware in a dynamic way, obtaining the actions performed in the victim system and providing means for the identification of malicious behavior. Furthermore it can overcome some shortcomings presented by other existing dynamic analysis approaches. The main contributions of BehEMOT are:

- The architecture of BehEMOT was designed to use a hybrid approach in the analysis environment (real and emulated) in such a way that it is possible to circumvent some techniques used by some malware to detect virtual machines and emulators;
- The monitoring of Windows native system calls performed by BehEMOT simplifies the process of capturing behaviors, producing both the functions and their arguments;
- The method for process monitoring used by BehEMOT makes it possible for all the children processes created by the malware to be analyzed in one single analysis. It also ensures that the system calls from other processes of the system, which are not related to the subject of the analysis, are not stored;
- The BehEMOT analysis doesn`t use intrusive techniques, thus preventing the malware from discovering that it is being monitored;
- Network activities are also captured, but outside the analysis environment. This way it is possible to obtain information from the behavior of *rootkits* that eventually use the network.

The remaining of this paper is organized as follows: Section 2 presents some concepts of dynamic analysis of malware, its advantages and disadvantages. In Section 3 the main approaches used in dynamic analysis are presented. Section 4 brings details of the implementation of BehEMOT and its components. In Section 5 the tests performed to validate the system and the results achieved are presented. Section 6 is composed by the final considerations about the system, with a brief comparison with other existing systems and future work.

2. MALICIOUS CODE ANALYSIS

Malicious code analysis is used to more deeply understand a malware operation—how it acts on the operating system, what kind of obfuscation techniques it uses, which of its execution flows leads to the main behavior, whether there is any network operation, especially file downloads, whether it captures information from the user and the system and so on.

Malware analysis can be divided into static and dynamic analysis. In the static case the behavior is extracted from the malware code without executing it, using techniques such as string analysis, disassembly and reverse engineering. Dynamic analysis requires the execution of the malware code, through debuggers, tools to monitor processes, registry and file operations or system call tracers.

Depending on the techniques and tools used during the analysis, the time spent may change, but in general static analyses are faster than dynamic ones. However, if reverse engineering is needed or if the malware sample has many execution flows or even if it is compressed with a hard-to-extract packer, then standard dynamic analysis is much faster and effective in order to provide significant results about the malware behavior. In [20] is presented a tool that can change a program in a way that obfuscates its execution flow, disguises its access to variables and makes it harder to follow register value changes, showing that the static analysis of a malware obfuscated by this tool is an NP-Hard problem. Moreover, during a static analysis, system responses to operations performed by the program cannot be obtained. In contraposition, dynamic analysis can be compromised by anti-analysis techniques used by some kinds of malware during its execution; also, only one of the paths that the code may execute can be observed.

3. TECHNIQUES FOR AUTOMATED DYNAMIC ANALYSIS

In this section we describe some current approaches used in dynamic analysis of malicious code, which are deployed by the main malware analysis system available. These systems focus on the analysis of PE-32 executable binaries in Windows systems; these follow the Windows file format that characterizes Windows executables and Dynamic Link

Libraries (DLL), the latter responsible for sharing functions. These files hold the assembly instructions that execute the actions that are performed in the system during the program execution, as well as some definitions necessary to its correct initialization [19].

The actions performed by a program consist of certain activities executed in the OS[1], such as opening a file, modifying registry entries, opening ports for network connections, creating processes etc. These activities represent the behavior of an executable file and so we can define the behavior of a malware as being the ordered set of actions executed in the OS and how the OS reacts to them. In order to obtain a malware behavior, we need to run it in an environment specially designed to monitor all of its important actions. There are some specific techniques for this purpose, and the most common are Virtual Machine Introspection, System Service Dispatch Table Hooking and Application Program Interface Hooking.

Virtual Machine Introspection (VMI) [11] is a kind of analysis which uses a virtual environment (guest system) to execute a malware while under monitoring of an intermediary layer between the virtual and real environment (host system), called Virtual Machine Monitor (VMM). This intermediary layer is responsible for obtaining and controlling the actions that are performed in the virtual environment, isolating it and minimizing the chances of the real environment being compromised. With this technique it is possible to obtain information from the low-level binary execution, such as the system calls that are performed and the state of memory and processor registers. However, the biggest restriction of this approach is the need for a virtual environment. Some kinds of malware which try to hide their actions can do specific tests to verify whether they are running over a virtual environment or not. If so, they change their behavior in order to hide their malicious execution flow [1].

The technique of System Service Dispatch Table (SSDT) Hooking, or capturing system calls (syscalls) at the kernel level, intercepts syscalls through hook functions, allowing for the modification of the execution flow [16] and of the answers returned to programs [12]. It is implemented through a module or driver, a special program that executes at kernel level (ring 0 in Windows OS) and is normally used as an interface between hardware devices and the user level (ring 3). A program operating at ring 0 has higher privileges, which permit capturing of all the syscalls made in the system. This technique can be applied to both virtual and real systems, since it only requires the installation of a driver in the system. Another advantage of using SSDT hooking is that it is not necessary to change the code of the file being monitored, avoiding detection by the integrity verification routines performed by some kinds of malicious software, in the quest to verify whether or not they are being analyzed. The drawback of applying this technique is its weakness to rootkits, a particular class of malware that executes at kernel level. In this case, as the malware is running at the same privilege level as the monitoring driver, it is possible for it to execute actions which bypass the monitor.

Another technique that can be used to capture malicious behavior is API hooking, in which the information capturing is done through modifications in the code of the executable being analyzed, in such a way that the API addresses used by the program are replaced by functions of the program responsible for the interception. When a malware is loaded into the system, all the DLLs it uses are verified, so that the interceptor can obtain the API addresses that it wants to modify [9]. After this step, the addresses are purposefully replaced. This technique can be easily subverted if the malware bypasses the API and performs the system calls directly on the system, or if it uses some DLL that is loaded during its execution. In these cases, the monitoring system will be unable to monitor the actions that are performed, losing a critical subset of the activities that belongs to the malware behavior. Another disadvantage is that the malware can easily detect if it is being monitored by checking whether there are any modifications to its own code.

For the purposes of this project, we chose the SSDT Hooking technique to deploy inside BehEMOT, mostly due to the fact that this approach does not require a virtual environment and because it does not make any changes in the code of the malicious software under analysis.

1 Operating System

4. SYSTEM DETAILS

In this section we describe BehEMOT's system architecture, detailing its internal components. As presented, BehEMOT is a behavioral malware analysis system with focus on binaries for the Windows OS. The data flow can be summarized as follows:

- A malware sample is inserted into the system through a control script, which verifies whether the malware is present in the malware database and has already been analyzed;
- The malware is executed in a Windows XP SP3 environment, installed with the default configuration, in addition to the BehEMOT components—a driver that applies the SSDT Hooking and its controller;
- The execution is limited to a configurable period of 4 minutes and the Windows XP environment is emulated by Qemu [4];
- The network traffic is captured by an external system which also hosts the control script;
- If there are no activities captured during the malware execution, or the execution stopped due to an error, the malware sample is automatically submitted to a similar environment in a real machine (without any emulation or virtualization);
- After the analysis, the control script receives the captured malware syscalls and network traffic, inserts them onto the database and processes the information gathered to generate a report with the behavior analysis.

In the next subsections, we detail the system architecture as a whole, the driver, its controller and the process of behavior analysis.

4.1 Architecture and Components

The BehEMOT tool was designed in a modular way, composed of the driver that implements the SSDT Hooking (more details on Subsection 4.2), the driver's controller (Subsection 4.3) and the analyzer (Subsection 4.4), the latter one responsible for interacting with the behavior analysis environment to obtain the malware actions and extract the behavior. This architecture is described in Figure 1. The complete environment is composed by a pool of emulated OSs using Qemu, a real OS (not emulated or virtualized), a firewall to capture the network traffic, isolate the systems and stop the attacks launched and a database, responsible for storing the malware samples and the analysis information.

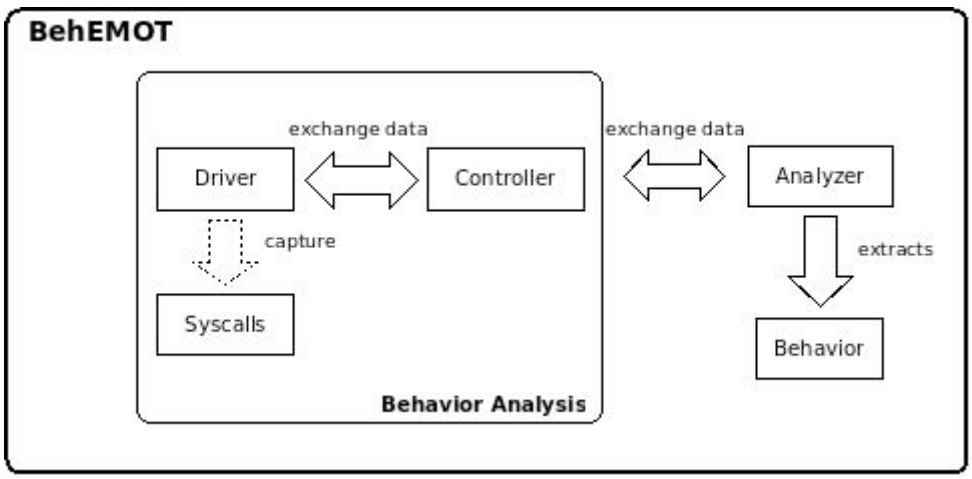

Figure 1. Behemot Architecture and its components.

Both the real and the emulated environments have the same configuration applied to the controller and the driver that compose BehEMOT. Also, as said before, the environments have a Windows XP with SP3 installed with the default configurations. No other application or additional tool, beyond the BehEMOT's components, is inserted into the environment, nor is made any kind of special configuration. This makes the systems similar to the ones found in fresh installations of the majority of Windows OS users. As explained previously, one of the environments is created with Qemu, an open source x86/ARM emulator that emulates all the devices of the system, such as the processor, which will

execute the assembly instructions of the malware. This environment is executed on the same machine as the BehEMOT analyzer, as an independent process, and all communication is made through a virtual network interface.

By using our environment we can do dynamic behavior analysis in an efficient way through the use of snapshots, a feature provided by Qemu in which the system is started from a previously saved state with the BehEMOT monitoring mechanisms already initialized, avoiding the time wasted in the initialization of the system. Another advantage is that it is easy to restore the entire system to a clean state after the tampering caused by the malware execution.

In the real analysis environment, an image of the clean system is generated in a separate partition used by the tool Partimage, which saves and restores states from a certain partition. This restoration process is done by a Linux OS, installed and loaded in another partition of this machine. This partition is activated whenever a behavior analysis is completed, restoring the victim OS to the known, previous state. The communication between the BehEMOT's analyzer and controller in the real environment is done through a network interface, on a local network.

4.2 Driver

The driver is the main component of the system, working at ring 0. It is responsible for capturing certain syscalls performed by processes through the interception of the System Service Dispatch Table (SSDT).

There are some Windows kernel structures which store information relative to the OS operation—process and thread, for example. One of these structures is the SSDT, a table where all the API memory addresses are located [6]. The SSDT is basically a vector of memory addresses, where each index corresponds to a system call. When a syscall needs to be executed, this table is queried in order to return the address of the appropriate function. These addresses are obtained through access to an exported kernel structure called KeServiceDescriptorTable. It can be accessed only by applications that execute at kernel level and consists of 4 fields, each one storing specific information. The fields relevant to the execution of the hook are the first—which contains a pointer to the beginning of the SSDT—and the third—which has the number of items present in the SSDT.

The SSDT hooking technique consists of the modification of the addresses present in the SSDT table by new ones that point to functions we control. In the driver initialization process, the hooking procedure starts by deactivating a protection that prevents memory write in the KeServiceDescriptorTable. To do this, the table is mapped through a Memory Descriptor List (MDL), a structure which describes a certain memory region and enables memory writing, turning changes in the addresses of SSDT possible.

For each SSDT modified address we need to associate a function, which points to the original syscall and is responsible for the interception or modification of the results received, as well as maintaining the correct behavior of the system. The original address contained in the SSDT must be stored before the change is made, allowing further use. The exchange of function addresses (old and new) is made in an atomic way by the driver.

When the driver initialization process is finished, the system is changed and every monitored syscall executed will pass through the functions defined in the driver. Since there are 391 system calls in the SSDT [6], we chose those that bring relevant information about the artifact execution [2] and can contribute in the definition of the malware behavior. These calls are related to file or registry operations, mutexes, processes and threads.

For each syscall chosen we create a new function that will execute the original call and store the parameters used, if the current running process is marked for monitoring. The choice of which syscalls must be monitored is based on the process identifier (PID) of the process who is performing it. The processes that need to be monitored are defined by the BehEMOT's controller and by the system calls that create new processes. The controller sends the PID of the malware process that needs to be analyzed to the driver through an I/O Request Packet (IRP). If, during said process execution the malware creates a new process, this new PID will be marked for monitoring too. When a marked process ends, its monitoring is also finished.

Besides the parameters, the source of the actions (performer) is stored too, making further analysis easier. The actions are reported as lines separated by the timestamp, performer, operation type and target fields, as shown in Table 1. The

collected information is periodically read by our controller through IRP. It then sends such information to the BehEMOT's analyzer, which will be detailed in a further section.

Table 1. Example of actions from the behavior of a malware sample, separated by field.

Timestamp	Performer	Operation type	Target
13:58:48.402	malware.exe	CREATE MUTEX	_AVIRA_21099
13:58:48.480	malware.exe	WRITE FILE	C:\WINDOWS\system32\sdra64.exe
13:58:48.965	malware.exe	OPEN PROCESS	C:\WINDOWS\system32\winlogon.exe

4.3 Controller

The communication between the driver and the external environment is done through the BehEMOT's controller, a program that executes in user level (ring 3). This controller is responsible for the creation of the malware process that is going to be analyzed, the initialization of the driver and the control and interpretation of the data provided by the BehHEMOT's driver during execution. After the controller is initializated, it will start the driver using the Service Control Manager (SCM). It is necessary to avoid pagination of the driver while it is running, so avoiding errors that are not handled by the kernel (BSOD - Blue Screen Of Death).

When the initial step is accomplished, the malware process is created in a suspended state, so that it is possible to fetch its PID and then forward it to the driver. Thus, the process that needs monitoring is identified and the execution is resumed, therefore initializing the malware execution. From this point on, the controller remains at a verification state, capturing the data sent by the driver, which will be formatted and stored in a file. Finally, after a timeout, the data file is closed and sent to the BehEMOT's analyzer.

4.4 Analyzer

The BehEMOT's analyzer is the component which guides the dynamic analysis process, sending the malware to the analysis environment and fetching the result after its execution. The analyzer decides whether the malware will be sent to the emulated or to the real machine, based on a previous execution that finished with error or with the detection of a packer. Packers cause problems when they are executed in environments based on Qemu, such as Armadillo, TE!Lock and ASProtect [1].

When the dynamic analysis ends, the analyzer processes the file containing the system calls, and generates a report with the extracted malware behavior at the OS level. This report also includes the network activities of the malware obtained by traffic capture, like file transfers, network scans, DNS queries and attacks in general.

5. TESTS AND RESULTS

For the tests we chose the kind of malware that indeed cause problems with similar analysis systems, i.e. those that are known to disrupt normal execution under a Qemu emulation environment. Among the malware samples that fit this requirement, we randomly chose a smaller subset to work with, except for those reference samples that had published results about them.

The goal of these tests was to validate the analysis system architecture as a whole, to evaluate the quality of the tool regarding the extraction of relevant behavior that can lead to the understanding of the malware activities in the victim system and to verify its effectiveness in the analysis of a greater variety of samples through the hybrid approach (emulated and real system). These verifications were made in two steps: first, we obtained a published analysis of a reference sample and compared it with the result from BehEMOT; second, by the submission of the chosen samples to BehEMOT and other systems publicly available that are well known and commonly used (Anubis and CWSandBox), and then comparing the results.

One of the chosen reference malware is Allaple, a polymorphic malware [22] whose predominant behavior is composed of the following critical activities: i) copy of its executable to %System%\urdvxc.exe; ii) modification of the registry key [HKCR\CLSID\{CLSID}\LocalServer32] to automatically execute after a system reboot [21]; iii) generation of ICMP traffic with a particular content; and iv) attempts to access several IP addresses through the NetBIOS protocol [22].

The result from executing the Allaple reference sample in BehEMOT, paired with the expected predominant behavior is shown in Table 2, which contains some pertinent snippets obtained from the BehHEMOT's analyzer.

In the presented case study, we obtained a behavior composed by 302 actions on the system until the termination of the main process (identified as malware.exe) and the created child-process (urdvxc.exe). In this behavior we noticed all the typical registry operations: read, creation and modification; also listed are files created, accessed and deleted by the processes, loaded libraries, network access, among other actions performed. With this, we could correlate the analyzed sample with a well known reference malware.

Table 2. Predominant behavior of a reference malware (worm Allaple) and an excerpt of the dynamically obtained behavior by BehEMOT.

Expected behavior	Behavior captured (BehEMOT)
%System%\urdvxc.exe	malware.exe;WRITEFILE;%System%\urdvxc.exe
HKCR\CLSID\{CLSID}\LocalServer32	urdvxc.exe;WRITEREGISTRY;HKCR\CLSID\{CLSID}\LocalServer32
ICMP TRAFFIC	16:54:22.099043 IP BEHEMOT > xxx.yy.aa.8: ICMP echo request, id 512, seq 1792, length 41 [...] 16:54:22.223208 IP BEHEMOT > xxx.yy.ee.23: ICMP echo request, id 512, seq 2816, length 41
NetBIOS TRAFFIC	16:54:27.073153 IP BEHEMOT.1060 > xxx.yy.ww.155.netbios-ssn: Flags [S] [...] 16:54:27.146635 IP BEHEMOT.1063 > xxx.yy.rrr.126.netbios-ssn: Flags [S]

To enhance the experiment, we did some execution tests with other malware samples, also submitted to the two analysis systems aforementioned that use different approaches to monitor malicious activities. In these tests, we selected samples of worms, trojans, viruses and bots, all of them identified by antivirus mechanisms. Table 3 shows the results achieved with these tests, in which we compared the predominant behavior that the malware should present with the activities captured by all three analysis environments. Among the determining factors we considered in the comparison are: i) the presence of network traffic; ii) specific, well-known actions on the victim system that indicate system compromise; and iii) non-interference in the normal operation of the analysis system, like identification of the monitoring environment, problems in the execution by limitation of the technology used and presence of “noise” resulting from activities not related with the malware execution.

We noticed that the CWSandbox reports usually show plenty of information that is not related to the malware behavior, such as processes responsible for controlling the analysis system itself, services that are native from the OS and update programs. This information can confuse the user, making him believe that these are actions executed by the malware during its analysis. Another problem is how easy it would be to detect the analysis environment, since the analysis identified several programs used in the analysis process. In the Anubis system, we found problems with some samples that caused the analysis to stop unexpectedly, mainly due to problems related to malware execution under a Qemu-based system. The behavior that was reported was incomplete, showing information just until the occurrence of the failure.

Table 3. Detected relevant behavior of malware dynamically analyzed by BehEMOT, Anubis and CWSandBox, where “Yes” indicates that malware monitoring finishes successfully, with useful information in the results, and “No” means that the malware did not execute as expected in the analysis environment, adopting techniques that made the analysis unfeasible and/or the report was too verbose about activities not considered to be malign, making the verification of the

sample behavior virtually impossible. The sample names were obtained by the ClamAV antivirus, whereas the packers, when identified, by PEFile [7].

Sample (Packer)	BehEMOT	Anubis	CWSandBox
Trojan.Agent (Armadillo v1.71)	Yes	No	No
Trojan.Agent (kkrunchy 0.23 alpha -> Ryd)	Yes	No	Yes
Worm.Allaple	Yes	Yes	Yes
Worm.Padobot	Yes	Yes	Yes
Worm.Palevo	Yes	No	No
Trojan.Buzus	Yes	Yes	No
Trojan.Ircbot (MEW 11 SE v1.2 -> Northfox[HCC])	Yes	Yes	No
Trojan.Sdbot	Yes	Yes	No
Trojan.Small	Yes	Yes	No
W32.Virut	Yes	Yes	Yes
PUA.Packed.tElock1.Private (tElock 0.98 -> tE!)	Yes	No	No

6. FINAL CONSIDERATIONS

Malware behavior analysis based on the monitoring of its actions in a target OS can generate useful information to improve the security of systems and networks. The benefits comprehend a deeper understanding of malware operation details, enhanced expertise to generate OS and application patches, ability to develop heuristic signatures for protection mechanisms and better security incident response. Thus, it is extremely important that the tool or environment that performs the malware analysis provide succinct and direct results, without being disguised by the report of commonly-found actions unrelated to the malign intent of the malware sample in the victim system.

However, we observe that a subset of the reports obtained from some malware analysis systems which are publicly available contains information that are not related to the actions executed by the malware itself (processess and applications executed normally by the OS), as well as activities related to the malware behavior monitoring process and to the control of the analysis environment. In other cases, we noted that there is not any filtering in the output data presented in the report. This incurs in irrelevant and repetitive information, which does not add any value to the analysis and may confuse the ordinary user. Our proposed tool, BehEMOT, treats those problems by directly monitoring the malicious process and its children process, and gathering a selected set of system calls relevant to malware behavior.

Thus, a report produced by BehEMOT is much more compact and understandable as compared to the Anubis and CWSandBox ones. The BehEMOT report contains specific information which explicits the ordered chain of actions executed by the malware and its children-process in the system, plus information referring to the network traffic, and so fully characterizing the sample behavior in the target system, as far as security issues are concerned. This also enhances the ability to develop effective countermeasures in cases of compromise by a specific, analyzed malware.

Another interesting point to notice is that some approaches that use only one kind of monitoring environment (emulation, virtualization or real system) suffer from very specific detection problems or from being subject to anti-analysis techniques and from the limitation of the method employed (emulators do not execute certain instructions, virtual machines are easily detected, real machines do not scale well and present hardware problems under overload conditions). As BehEMOT uses a hybrid approach mixing emulation and real system, this problem is minimized, generating a greater flow in the amount of analyses completed without errors.

One thing that should also be noticed is that all dynamic analysis approaches that require a driver being inserted into the monitored OS to capture information may present problems when running in parallel with rootkits. Depending on how the rootkit is installed, it can conflict with the monitoring driver and so cause kernel errors, thus avoiding the analysis to be completed. This kind of situation can be circumvented only if the environment analysis is being externally monitored, since the analysis in a real environment is unable to complete. Another problem with rootkits monitoring happens due to their usually not having a PID, as drivers normally do. In this case, BehEMOT will be able to monitor only the process that starts the rootkit, which will generate an incomplete analysis. On the other hand, the network traffic capture occurs

externally to the analysis environment and will aggregate useful information to the report, thus allowing the identification of some malicious activities.

Although the mixed approach handles some problems related to anti-analysis techniques, if the amount of malware using these techniques becomes far superior than of those that can execute in an emulated environment, the same disadvantages regarding the intensive use of real machines will occur. Besides, the actual analysis is limited to binaries for Windows XP, discarding those i) which execute solely on other Windows versions; ii) which execute on other OSs; iii) intended for hardware platforms other than the more common x86; iv) niche applications that run on dedicated software environments and that can present malicious behavior, like PDF viewers, Javascript scripts or Flash plugins.

To sum up, although we have not yet tested the environment in an exhaustive way, with hundreds of thousands of samples available, the diversity of tested samples points to promising results, and the environment can be extended to deal with different types of files. As follow-up work, we are currently i) generating behavior trees obtained from data obtained by BehEMOT; ii) scoring individual actions to help detect malicious behavior; iii) performing behavior analysis of PDF files; iv) supervising malicious activities performed by web browsers and v) elaborating countermeasures on feedback from BehEMOT analyses to mitigate the effect of compromised systems.

REFERENCES

[1] Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E. and Vigna, G., "Efficient detection of split personalities in malware", 17th Annual Network and Distributed System Security Symposium (2010).

[2] Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., and Kruegel, C., "A View on Current Malware Behaviors", Usenix Workshop on Large-scale Exploits and Emergent Threats (LEET) (2009).

[3] Bayer, U., Kruegel, C. and Kirda, E., "TTanalyze: A Tool for Analyzing Malware", Proc. 15th Ann. Conf. European Inst. for Computer Antivirus Research (EICAR), 180-192 (2006).

1. Bellard, F., "QEMU, a fast and portable dynamic translator", Proc. of the Annual Conference on USENIX Annual Technical Conference, USENIX Association, 41-41 (2005).

[4] Binsalleeh, H., Ormerod, T., Boukhtouta, A., Sinha, P., Youssef, A., Debbabi, M. and Wang, L., "On the Analysis of the Zeus Botnet Crimeware Toolkit", Proc. of the Eighth Annual Conference on Privacy, Security and Trust, PST'2010 (2010).

[5] Blunden, B., [The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System], Jones and Bartlett Publishers, Inc, 1th edition (2009).

[6] Choi, Y., Kim, I., Oh, J. and Ryou, J., "PE File Header Analysis-Based Packed PE File Detection Technique (PHAD)", Proc of the International Symposium on Computer Science and its Applications, 28-31 (2008).

[7] Dinaburg, A., Royal, P., Sharif, M., and Lee, W., "Ether: Malware analysis via hardware virtualization extensions", Proc. Proceedings of The 15th ACM Conference on Computer and Communications Security (CCS 2008), October (2008).

[8] Father, H., "Hooking Windows API-Technics of Hooking API Functions on Windows", CodeBreakers J., vol.1, no.2 (2004).

[9] Franklin, J., Paxson, V., Perrig, A. and Savage, S., "An Inquiry Into the Nature and Causes of the Wealth of Internet Miscreants", In Conference on Computer and Communications Security (CCS) (2007).

[10] Garfinkel, T. and Rosenblum, M., "A virtual machine introspection based architecture for intrusion detection", Proc. Network and Distributed Systems Security Symposium, 191-206 (2003).

[11] Hoglund, G. and Butler, J., [Rootkits: Subverting the Windows Kernel], Addison- Wesley Professional, 1th edition (2005).

[12] Holz, T., Engelberth, M. and Freiling, F., "Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones", Reihe Informatik TR-2008-006, University of Mannheim (2008).

[13] http://www.joebox.org/

[14] Kang, M. G., Poosankam, P., and Yin, H., "Renovo: A hidden code extractor for packed exe-cutables", Proc. of the 2007 ACM Workshop on Recurring Malcode (WORM 2007) (2007).

[15] Kong, J. [Designing BSD Rootkits], No Starch Press, 1th edition (2007).

[16] Leder, F. and Werner, T., “Know your enemy: Containing conficker”, The Honeynet Project & Research Alliance (2009).
[17] Martignoni, L., Christodorescu, M., and Jha, S., “Omniunpack: Fast, generic, and safe unpack-ing of malware”, Proc. of the Annual Computer Security Applications Conference (ACSAC) (2007).
[18] http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff_v8.docx
[19] Moser, A., Kruegel, C., and Kirda, E., “Limits of Static Analysis for Malware Detection”, In ACSAC, 421-430. IEEE Computer Society (2007).
[20] http://www.securelist.com/en/descriptions/old145521
[21] http://www.softpanorama.org/Malware/Malware_defense_history/Malware_gallery/Network_worms/allaple_rahack.shtml
[22] Song , D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P. and Saxena, P., “BitBlaze: A New Approach to Computer Security via Binary Analysis”, Proc. of the 4th International Conference on Information Systems Security (2008).
[23] Willems, C., Holz, T. and Freiling, F., "Toward Automated Dynamic Malware Analysis Using CWSandbox", IEEE Security and Privacy, vol. 5, no. 2, 32-39, (2007).
Yegneswaran, V., Saidi, H., Porras, P., “Eureka: A framework for enabling static analysis on malware”, Technical Report SRI-CSL-08-01 Computer Science Laboratory and College of Computing, Georgia Institute of Technology (2008).