Reading the contents of deleted and modified files in the virtualization based black-box binary analysis system Drakvuf

S.G. Kovalev <skovalev@ptsecurity.com>
Positive Technologies
8 Preobrazhenskaya Square, Moscow, 107061, Russia

Abstract. The article discusses ways to get the content of files, which are modified during the processing in the well-known open source dynamic analysis environment Drakvuf. Drakvuf initially implemented file saving functionality based on the use of undocumented mechanisms for working with the system cache. The author of this article proposes a new approach to obtaining the content of files on Microsoft Windows family systems using Drakvuf. The proposed approach is based solely on the use of the public interface of the kernel by the hypervisor and provides portability between different versions of the operating system. In the conclusion of the article, the advantages and disadvantages of both approaches are presented, and directions for further work are proposed.

Keywords: malware; dynamic analysis; injection; Drakvuf; Virtual Machine Introspection.

DOI: 10.15514/ISPRAS-2016-30(5)-7

For citation: Kovalev S.G. Reading the contents of deleted and modified files in the virtualization based black-box binary analysis system Drakvuf. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018. pp. 109-122. DOI: 10.15514/ISPRAS-2018-30(5)-7

1. Introduction

In recent years, a steady increase in the number of malicious programs has been registered [1]. A direct consequence was the impossibility of manual analysis of this thread, which led to the emergence of the need for scalable and automated tools for collection and analysis of malware. Such tools include honeypots [2, 3] and sandboxes [4, 5]. At the same time, it is worth noting that malware uses various techniques to detect analysis tools [6], which imposes large restrictions on such tools.

The use of virtual machine monitors provides several advantages for creating such tools: isolation of a program of interest, the ability to quickly and easily restore a compromised system, as well as scalability. Dynamic analysis requires the completeness and accuracy of collected data. The use of virtual environments also allows meeting these requirements, providing an analysis environment with information about code execution, disk and memory usage in real time.

There are at least two approaches to the use of virtualized environments for dynamic analysis. In the first case, the virtual machine is used as a replacement for the physical one, which allows for scalability and a greater speed of restoring the analysis environment; wherein, the kernel driver or injection of DLL into the address space of a process of interest is used for the analysis. In the second case, the virtual machine monitor is expanded with new possibilities of studying the virtual machine state and does not require the installation of additional software in the system of interest. This approach is called "virtual machine introspection" [7] or VMI.

One of the important components of dynamic analysis is reading the contents of deleted and modified files. This is due to the fact that some malicious programs download the payload over the network and save it in a temporary file. This class of malware was named Trojan Downloader [8]. Reading the contents of a newly created file is necessary for further analysis. Another class of malware, called Ransomware Trojans [9], encrypts many files on the disk. The presence of information about a large number of newly created files with similar names or about modified contents is a necessary condition for detecting malicious behavior.

Further, this article discusses the dynamic analysis environment Drakvuf [10, 11], and two ways to read the contents of files. The first method was present initially and was built on the knowledge of internal structures of the operating system kernel. The second method was added by the author of this article and is based on the injection of system functions. Thus, this approach relies on the stable public API of the operating system kernel and in some cases allows reading the full contents of files.

2. Overview of the dynamic analysis environment Drakvuf

Drakvuf is a virtualization based agentless black-box binary analysis system based on «virtual machine introspection».

To build this environment, the following solutions were used:

- Xen virtual machine monitor [12];
- LibVMI library [13], which allows access to the low-level state of a virtual machine;
- Rekall framework for studying the virtual memory of an operating system of interest [14].

Further, each of the components and the way to use them together with Drakvuf are discussed.

2.1. Xen

Xen is a bare-metal (i.e. independent of the operating system) hypervisor that supports hardware virtualization technology. Xen allows running multiple virtual machines, the so-called DomU. In this case, one of them is considered to be controlling, the so-called Dom0.

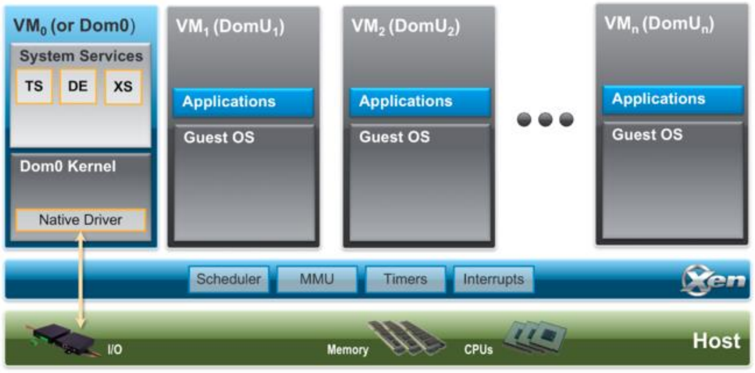

Fig. 1. Xen architecture

Xen provides resource allocation for virtual machines, scheduling of virtual kernels, and interrupt control. Dom0 is used to interact with the user, providing the system with external device drivers (NIC, SATA, etc.). Dom0 typically runs the QEMU process [15] associated with each virtual machine. QEMU provides emulation of a virtual machine target platform (system logic set, BIOS or UEFI, external devices). QEMU execution is supported in dedicated domains, the so-called subdomains, which increases safety and performance.

Starting from Xen 4.5, the API for VMI is added to the hypervisor. Subsequently, this interface is constantly being improved.

2.2. LibVMI

LibVMI is a library that provides access to the state of a virtual machine. It provides the following capabilities (the list is not complete):

- reading and changing the contents of the virtual memory of a VM of interest;
- setting permissions to the physical memory of a VM;
- reading and changing the values of VM processor registers;
- stopping and resuming VM operation;
- installing handlers for certain hardware events in a VM:
    - changing values of control registers (CR0, CR3, CR4);
    - access violations to the physical memory of a VM;
    - single-step debugging of the VM execution;
    - debugging interrupt (INT 3).
- LibVMI uses Xen VMI API for hidden analysis and change of the VM state.

2.3. Rekall

Rekall is a virtual memory analysis framework. In the context of Drakvuf, the feature of building an operating system profile based on debugging symbols of a specific operating system is promising.

For example, for operating systems of the Microsoft Windows family, symbols of the kernel and main modules are provided in the PDB format. Rekall allows converting a PDB file to the JSON format. Such JSON file is called a profile [16] and contains the following information:

- a brief description of the kernel for which the profile has been compiled (family, version, build number);
- a list of constants and their offsets in the kernel;
- description of structures (names of members and their offsets within structures).

The presence of such information allows overcoming the semantic gap between the analysis environment and a system of interest.

2.4. Drakvuf

Drakvuf combines the ability to analyze and change the state of the VM provided by LibVMI and the debugging information provided by Rekall with the knowledge of the internal structure of an operating system of interest. This allows achieving the following features:

- detection of the current process and thread at an arbitrary point in time;
- detection of the virtual address of a symbol (of constant or function) by name;
- setting a virtual address trap;
- getting the file name according to the file handle.
- In addition, Drakvuf provides a plugin architecture and an initial set of plugins. Plugins include the following ones:
- syscalls – allows tracking entry points to system call handlers;
- filedelete – allows reading the contents of deleted and modified files.
- In the presented work, the filedelete plugin has been significantly improved, as will be discussed in section 5.
- It is important to note that all useful activities are performed during the processing of exit from a VM (the so-called “VM exit”). Thus, the Drakvuf operation scheme is as follows:
- at the very beginning of Drakvuf operation, the VM is stopped;
- traps and event handlers are configured (in plugins);
- the main loop is started:
    - VM operation is resumed, and Drakvuf begins to wait for notification of an event;
    - one of the expected events occurs in the VM;

- Xen stops the VM and transfers control to Dom0, where Drakvuf is usually running;
- Drakvuf (LibVMI) bypasses the list of handlers for events of this type, transferring the control to each of them by rotation.

2.5. Using the Drakvuf trap mechanism to determine deleted and changed files

To determine deleted and changed files, traps on the following system functions are installed from ntoskrnl.exe:

- NtSetInformationFile – is used to delete a file when closing the last file handle;
- NtWriteFile – records data to a file;
- NtClose – closes the file handle.

The NtWriteFile handler adds the following data to the list: PID of the process, file handle, and file name. The NtClose handler for modified files removes an entry from the list and proceeds to reading the contents of a file. The NtSetInformationFile handler proceeds to read the contents of any deleted file.

3. Reading the contents of files by analyzing the cache manager

For a detailed presentation of the material see [17]. The following is a general description of the approach which is necessary and sufficient for comparison.

In the beginning, the _FILE_OBJECT structure location is determined according to the file handle. Next, using the value of a member of the SectionObjectPointer structure, the location of the _SECTION_OBJECT_POINTERS structure is determined, the DataSectionObject member of which points to the _CONTROL_AREA structure. At the end of this structure, there is the first member of a linked list, consisting of _SUBSECTION structures. Each such structure defines a sequential memory chunk mapped to a file. Having read the contents of all such chunks, one can compile a file (or at least part of it, see below).

In the _SUBSECTION structure, the following members are significant:

- SubsectionBase – the first member of the array of _MMPTE entries, each of which defines the physical address of a page (in terms of VM) and some flags;
- PtesInSubsection – the number of array members;
- StartingSector – the offset of the first page of this section in the file, expressed in chunks of 512 bytes.

Each _MMPTE entry is a 4 KB virtual memory page descriptor (the so-called PTE, or “page table entry”). Collectively, PTEs describe a continuous virtual memory block that represents a portion of the file starting from the StartingSector*512 offset. However, some pages can be paged out from the RAM of the VM. This is indicated by the zero value of the Present flag in PTE.

Thus, in order to read the cache manager contents, Drakvuf just needs to bypass the list of PTEs for each section and to read the contents of each page for which the Present flag is set.

3.1. Limitations of this approach

Although this approach provides reading of the cache manager content in a way that is fast and invisible for the VM, it has several limitations:

- large files can be accessed in parts; in this case, the cache manager may contain one or more fragments of the file, while the rest of the file even will not be downloaded;
- the current implementation does not take into account the fact that some pages with cleared Present flag may still contain data not downloaded to disk;
- memory for cache manager structures is allocated from the system working set and can be paged out to disk;
- the current implementation does not support working with memory-mapped files [18].

The above limitations led to the beginning of work on injection of system functions to read the contents of files.

4. Injection of system calls

Initially, the linbinjector library was added to Drakvuf, which provided an injection of the CreateProcess system function. This allowed for the direct launch of an application of interest in the VM, requiring only the presence of a file on the VM disk. This approach (the so-called agent-free approach) provides greater secrecy compared to the classical solution in which the remote control process was launched in the VM. Since the injection of functions is an integral part of the proposed solution, here is a general description of the approach.

The function injection implies a change in the state of the current instruction stack and register (IP on x86 architecture), which emulates the sequence of operations used by the compiler when calling a function (the word “call” can be further used instead of the word “injection”).

Since operating systems of the Microsoft Windows family are considered, the rules for calling functions in the kernel are well documented [19]. For example, let us consider the injection of the ZwQueryVolumeInformationFile function call on a 64-bit system.

This function takes five arguments: object handle (integer), pointer to the IO_STATUS_BLOCK structure, pointer to the FILE_FS_DEVICE_INFORMATION out structure (for the example), size of out structure, structure type (integer, for FILE_FS_DEVICE_INFORMATION is 4). In accordance with the accepted ABI, the first four arguments are transferred in RCX, RDX, R8, R9 registers, and the last argument is transferred on the stack.

However, there are some limitations that shall be considered:

- before calling a function on the stack, space for four arguments is reserved (the so-called “home space”);
- when transferring a pointer to a structure, the address of the structure beginning must be aligned with a value equal to the greatest alignment of any member of the structure;
- before calling a function, the stack shall be aligned by a multiple of 16 B.

The last two requirements were not initially taken into account, which led to a time-consuming debugging of various fatal kernel errors (the so-called BSOD).

After all arguments of the function are prepared, the return address is set on the stack. As a rule, it coincides with the trap address, which allows continuing execution of the VM. In this case, the trap is not deleted, which is necessary for processing the exit from ZwQueryVolumeInformationFile.

Lastly, the ZwQueryVolumeInformationFile address is entered to the RIP register and the VM operation is resumed.

Since it is possible to setup new trap, after the ZwQueryVolumeInformationFile function completes, the trap handler receives control again, which allows processing returned data, restoring registers and the stack, and continuing operation of the VM.

Further development of this approach led to the idea of the possibility of sequential execution of several injections, which allowed reading the contents of files without reference to the cache manager structures.

5. New approach to reading the contents of files by injection of system calls

The proposed approach was a direct consequence of the desire to achieve guaranteed reading of the contents of arbitrarily large files, not limited to what is contained in the cache manager. The kernel already provides the ZwReadFile system function. However, one cannot simply call ZwReadFile on the handle of an arbitrary object:

- the handle can be linked with a logical disk volume, I/O device, etc.;
- to read files, one needs to prepare a memory buffer of sufficient size;
- for files that do not fit into the buffer, several read operation calls are required;
- reading of asynchronous files can lead to unexpected errors.

In the course of the work, the author discovered at least two more circumstances that were not initially taken into account:

The stack size in the kernel mode is limited (16 KB for 32-bit systems and 24 KB for 64-bit systems), so it is impossible to reliably allocate a sufficiently large memory buffer on the stack;

in a multithreading OS, a process or a thread may switch while the contents of the file are being read;

in LibVMI, all trap handlers registered to a virtual address are sequentially called, so it is necessary to distinguish the beginning of the chain from its middle.

Here is the brief description of solutions for each of these limitations.

5.1. Solving the problem of determining the type of the handle

The ZwQueryVolumeInformationFile function called with the FileFsDeviceInformation parameter returns the FILE_FS_DEVICE_INFORMATION structure. The first member of this structure DeviceType takes one of the values [20]. During the research, it was revealed that regular files are of FILE_DEVICE_DISK type (i.e., 0x7).

5.2. Solving the problem of buffer preparation

In the beginning of work, the author did not take into account the fact, that the kernel stack size is not only limited but also rather small (16 kB for 32-bit systems and 24 kB for 64-bit systems). Thus, in the first version, the 4kB buffer was allocated directly on the stack. However, the author has soon noticed that in some cases OS has a fatal error when reading a file. It was suggested that the reason is a kernel stack overflow.

In order to eliminate such an error, it was decided to allocate the buffer in a non-paged memory area (so-called «NonPaged Pool»). This provides an additional advantage of the possibility to allocate more memory (for example, 64 KB).

For further optimization, the allocation of new memory buffer on request was added. All allocated memory buffers are put in the list. Initially, the list is empty. Each new thread first accesses the list. If there is a free memory buffer in the list, it is marked as busy and used for reading operation. If there are no free buffers in the list, the ExAllocatePoolWithTag function is called (injected) first.

In practice, it turned out that a single memory buffer is sufficient for a VM with two kernels.

5.3. Solving the problem of reading large files

In practice, there are often large files that do not fit in one memory buffer. Therefore, it becomes necessary to perform the file read operation in a loop. However, the file size is not known in advance. It would be possible to use one of the system functions to read the file size, but this would extend the call chain and reduce system performance. In addition, there is a need to move the carriage in the file. Fortunately, the ZwReadFile function already has all the necessary properties to solve this problem.

One of the ZwReadFile arguments is a pointer to the IO_STATUS_BLOCK structure. Upon the completion of the read operation, this structure contains two members: the operation completion code and the number of bytes read.

The second useful argument in the context of this task is the ZwReadFile argument, which is a pointer to the LARGE_INTEGER ByteOffset structure. This argument allows setting the offset in the file from which ZwReadFile will start reading the contents.

Using the second member IO_STATUS_BLOCK and ByteOffset allows creating a simple read algorithm for a large file: as long as the read operation returns STATUS_SUCCESS and the number of bytes read is equal to the size of the transferred memory buffer, continuing the read operation, increasing the offset by the memory buffer size. Wherein, at the beginning it is necessary to explicitly specify a zero offset, because in practice, at the time of calling NtClose, the carriage was shifted to the end of the file. This resulted in a read error STATUS_END_OF_FILE.

5.4. Solving the problem of asynchronous files

At the beginning of the research, it was noticed that ZwReadFile often returns the STATUS_PENDING error code. This means that an attempt to read a file opened for asynchronous access is being made [18]. The first solution was to add a call to the WaitForSingleObject function. This call is different from others. There was the need to keep the stack from the previous call ZwReadFile and the lack of its own handler. The only thing that the trap handler did on WaitForSingleObject was transferring control to the ZwReadFile handler, which again checked the error code and read the memory buffer.

However, it soon became clear that the operation of the system became unstable. Often there were fatal kernel errors associated with breaking the stack. Debugging of the kernel showed that in almost all cases the stack pointer was more than 1 MB from the base of the nuclear stack (so-called “stack underflow”). A further study of the stack showed that the violation of the stack began with calling ZwReadFile. It was not possible to establish the exact cause of the error, but there was a clear dependence of the error reproducibility on the type of files read. Errors were reproduced when accessing asynchronous files.

Thus, it was decided not to attempt to read such files. Finding out whether the file was open for asynchronous access turned out to be trivial. The _FILE_OBJECT structure contains the Flags member. If the FO_SYNCHRONOUS_IO flag is set, the file has been opened for synchronous access. So it is possible to read its contents.

This simple revision led to an increase in the reliability of the entire system. However, the issue of reading the contents of files opened for asynchronous access remained open. The answer to this question is partly given below.

5.5. Solving the problem of processing several traps at one virtual address

The need to process the returned values of called functions results in at least two handlers at one virtual address: a constant handler at NtClose and a temporary handler for the function being called. The situation is aggravated by the fact that the handler of each stream can be installed to the same address. Thus, it is necessary not only to distinguish the beginning of a call chain but also to distinguish between processes and threads. Moreover, it turned out that all handlers of traps to a given virtual address are traversed in LibVMI. Therefore, upon completion of the read operation, one cannot simply delete a trap. This will lead to looping attempts to read the file.

To solve this problem, a map was added, which maps a pair of process and thread values to a marker for completion of a read operation. When a trap on NtClose is triggered and a decision is made to read the contents of a file, a new process thread pair is added to the map with an empty marker. When a file read operation is completed in any form, a marker in the map for the current process thread is filled, and the trap is deleted. Since in LibVMI new traps are added to the top of the list, for the current process thread a trap on NtClose is executed the last. It checks the marker and, if it is full, the entry is deleted from the map, and the handler ends.

At the same time, the handler of each called function checks the compliance of the current process-thread with the stored value, which eliminates the accidental triggering of the handler.

5.6. Solution algorithm

By putting together all of the above, the following file reading algorithm is obtained:

- Step 1. Check that the file is open for synchronous access, otherwise shut down.
- Step 2. Check that no read operations are performed for the current process-thread and add a marker to the map, otherwise remove the marker from the map.
- Step 3. Call ZwQueryVolumeInformationFile and check that the regular file is processed, otherwise fill in the marker and complete the work.
- Step 4. Allocate a memory buffer if there is a free one, otherwise call ExAllocatePoolWithTag.
- Step 5. In the loop, call ZwReadFile as long as the error code is STATUS_SUCCESS and the number of bytes read is equal to the size of the memory buffer.
- Step 6. Fill in the marker for the current process thread.

If one of the steps fails by mistake, the attempt to read the file is considered failed, and an attempt to read parts of the file from the cache manager is made. This partly solves the problem with asynchronous files.

Thus, the proposed approach significantly expanded the existing one, allowing reading the contents of files reliably, using the documented system functions.

6. Conclusion

The paper presents a new approach for reading the contents of deleted and modified files during automated dynamic analysis of applications on Microsoft Windows operating systems. This approach has a distinctive feature of using the mechanism for injecting system functions of the operating system running in a virtual machine from the side of the hypervisor. This technique avoids the presence of agent applications or drivers in the virtual machine and increases secrecy, which is extremely important in studying the malware. It uses documented system functions, which allows achieving transferability between different versions of operating systems of this family.

The problem of reading the contents of files opened for asynchronous access is not fully solved, which sets the direction for further activities.

In addition, this paper provides an overview of the dynamic analysis environment Drakvuf, its constituent parts and some principles of work. It considers the initial approach to reading the contents of files based on reading internal structures of the cache manager, and its limitations.

References

[1]. The Independent IT-Security Institute. Malware. Available at: https://www.av-test.org/en/statistics/malware/, accessed 17.11.2018.

[2]. Asrigo K., Litty L., Lie D. Using VMM-Based Sensors to Monitor Honeypots. Department of Electrical and Computer Engineering University of Toronto, 2006. Available at: https://security.csl.toronto.edu/papers/asrigo-vee2006.pdf, accessed 17.11.2018.

[3]. Rangian M.K., Attri U. Design and Implementation of Malware Collection System Based on Client Honeypot. International Journal of Scientific & Engineering Research, vol. 4, issue 3, 2013, pp. 775-780.

[4]. Cuckoo Sandbox. Available at: https://cuckoosandbox.org/, accessed 17.11.2018.

[5]. Willems C., Holz T., Freiling F. Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Security & Privacy, vol. 5, issue 2, 2007, pp. 32-39.

[6]. Malware Anti-Analysis Techniques and Ways to Bypass Them. Available at: https://resources.infosecinstitute.com/malware-anti-analysis-techniques-ways-bypass/, accessed 02.05.2017.

[7]. Garfinkel T., Rosenblum M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. Computer Science Department, Stanford University, 2003. Available at: https://suif.stanford.edu/papers/vmi-ndss03.pdf, accessed 17.11.2018.

[8]. Kaspersky Lab. Malware Classification (in Russian). Available at: https://www.kaspersky.ru/blog/klassifikaciya-vredonosnyx-programm/2200/, accessed 17.11.2018.

[9]. Symantec Corporation. What Is Ransomware? Available at: https://us.norton.com/internetsecurity-malware-ransomware.html, accessed 17.11.2018.

[10]. Drakvuf. Available at: https://drakvuf.com/, accessed 17.11.2018.

[11]. Lengyel T.K. Malware Collection and Analysis via Hardware Virtualization. University of Connecticut, 2015. Available at: https://tklengyel.com/thesis.pdf, accessed 17.11.2018.

[12]. Xen Project. Available at: https://xenproject.org/, accessed 17.11.2018.
[13]. LibVMI. Available at: http://libvmi.com/, accessed 17.11.2018.
[14]. Rekall Forensics. Available at: http://www.rekall-forensic.com/, accessed 17.11.2018.
[15]. QEMU. Available at: https://www.qemu.org/, accessed 17.11.2018.
[16]. Rekall Profiles. Available at: http://blog.rekall-forensic.com/2014/02/rekall-profiles.html, accessed 17.11.2018.
[17]. Russinovich M., Solomon D., Ionescu A. Microsoft Windows Internal Design. The Main OS Subsystems, 6th ed. (in Russian). Saint Petersburg, Piter, 2014, 672 p.
[18]. Richter J., Nazar C. Windows via C/C++. Visual C++ Programming (in Russian). Saint Petersburg, Piter, 2009, 896 p.
[19]. Building C/C++ Programs. Available at: https://docs.microsoft.com/en-us/cpp/build/building-c-cpp-programs?view=vs-2017, accessed 17.11.2018.
[20]. Specifying Device Types. Available at: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/specifying-device-types, accessed 17.11.2018.

Получение содержимого удаляемых и изменяемых файлов в среде динамического анализа исполняемых файлов Drakvuf

С.Г. Ковалёв <skovalev@ptsecurity.com>
Positive Technologies
107061, Москва, Преображенская пл., д. 8

Аннотация. В статье рассматриваются способы получения содержимого файлов, изменяемых в процессе работы известной среды динамического анализа с открытым исходным кодом Drakvuf. В Drakvuf изначально реализована функциональность сохранения файлов, основанная на использовании недокументированных механизмов работы с системным кэшем. Автором данной статьи предложен новый подход получения содержимого файлов в системах семейства Microsoft Windows с помощью Drakvuf. Предложенный подход основан исключительно на использовании публичного интерфейса ядра со стороны гипервизора и обеспечивает переносимость между различными версиями операционной системы. В завершение статьи приведены достоинства и недостатки обоих подходов, предложены направления дальнейших работ.

Keywords: вредоносная программа; динамический анализ; инъекция; Drakvuf; Virtual Machine Introspection.

DOI: 10.15514/ISPRAS-2018-30(5)-7

Для цитирования: Ковалёв С.Г. Получение содержимого удаляемых и изменяемых файлов в среде динамического анализа исполняемых файлов Drakvuf. Труды ИСП РАН, том 30, вып. 5, 2018 г., стр. 109-122 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(5)-7

Список литературы

[1]. Malware. The Independent IT-Security Institute. Доступно по ссылке: https://www.av-test.org/en/statistics/malware/.

[2]. Kurniadi Asrigo, Lionel Litty, David Lie. Using VMM-Based Sensors to Monitor Honeypots. Department of Electrical and Computer Engineering University of Toronto, 2006. Доступно по ссылке: https://security.csl.toronto.edu/papers/asrigo-vee2006.pdf, дата обращения 17.11.2018.

[3]. Manpreet Kaur Rangian, Upasna Attri. Design and Implementation of Malware Collection System Based on Client Honeypot. International Journal of Scientific & Engineering Research, 2013.

[4]. Cuckoo Sandbox. Доступно по ссылке: https://cuckoosandbox.org/, дата обращения 17.11.2018.

[5]. Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. Security & Privacy, IEEE, 2007.

[6]. Malware Anti-Analysis Techniques and Ways to Bypass Them. Доступно по ссылке: https://resources.infosecinstitute.com/malware-anti-analysis-techniques-ways-bypass/, дата обращения 02.05.2017.

[7]. Tal Garfinkel, Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. Computer Science Department, Stanford University, 2003. Доступно по ссылке: https://suif.stanford.edu/papers/vmi-ndss03.pdf, дата обращения 17.11.2018.

[8]. Классификация вредоносных программ. Доступно по ссылке: https://www.kaspersky.ru/blog/klassifikaciya-vredonosnyx-programm/2200/.

[9]. What is ransomware?. Доступно по ссылке: https://us.norton.com/internetsecurity-malware-ransomware.html, дата обращения 17.11.2018.

[10]. Drakvuf. Доступно по ссылке: https://drakvuf.com/, дата обращения 17.11.2018.

[11]. Tamas Kristof Lengyel. Malware Collection and Analysis via Hardware Virtualization. University of Connecticut, 2015. Доступно по ссылке: https://tklengyel.com/thesis.pdf, дата обращения 17.11.2018.

[12]. Xen Project. Доступно по ссылке: https://xenproject.org/, дата обращения 17.11.2018.

[13]. LibVMI. Доступно по ссылке: http://libvmi.com/, дата обращения 17.11.2018.

[14]. Recall Forensics. Доступно по ссылке: http://www.rekall-forensic.com/, дата обращения 17.11.2018.

[15]. QEMU. Доступно по ссылке: https://www.qemu.org/, дата обращения 17.11.2018.

[16]. Rekall Profiles. Доступно по ссылке: http://blog.rekall-forensic.com/2014/02/rekall-profiles.html, дата обращения 17.11.2018.

[17]. М. Руссинович, Д. Соломон, А. Ионеску. Внутреннее устройство Microsoft Windows. 6-е издание. Основные подсистемы ОС. СПб.: Питер, 2014, 672 с.

[18]. Джеффри Рихтер, Кристоф Назар. Windows via C/C++. Программирование на языке Visual C++. СПб.: Питер, 2009, 896 с.

[19]. Building C/C++ Programs. Доступно по ссылке: https://docs.microsoft.com/en-us/cpp/build/building-c-cpp-programs?view=vs-2017, дата обращения 17.11.2018.

[20]. Specifying Device Types. Доступно по ссылке: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/specifying-device-types, дата обращения 17.11.2018.