Malware Analysis

Maarten van Elsas Arris Huijgen

February 29, 2016

1 Introduction

In 20 years, the Internet has grown from a network connecting a few academic institutions among eachother to a network to which everyone is connected with multiple devices. Besides the social impact of the Internet, it has also become an indispensable part of our economic system. This also comes with the negative consequences of people attempting to maliciously use it for their own benefits. Malicous software, shortened to *malware* supports these people to accomplish their goals.
This paper is focussed at providing an overview of the ways to perform analysis. As Microsoft Windows currently the most prevalent Operating System (OS) and therefore, most malware has been written for this OS, this paper will primarily discuss Windows-based malware, although various of the concepts will also apply for malware that runs on other OSs.
First, some general information about malware and ways of analyzing malware will be provided. Second, various approaches for collecting information about the malware's behavior will be discussed. In section thereafter, the ways in which an analysis environment can be setup are explained, including the possible complicating factors implemented in the malware. In the second part of this paper, an approach to automatically capture and reverse engineer network traffic will be explained. Lastly, a method to automatically classify malware will be discussed.

2 Malware analysis

When analyzing malware, various categories of malware can be identified. Moreover, different approaches of analyzing malware are available. Both the types of malware and the approaches of analyzing malware will be explained in the following paragraphs.

2.1 Malware Types

Malware can be distinguished in various different types which are listed below [1][6].

- **Logic Bomb**: Uses a trigger to execute a certain payload. The condition of the trigger is only limited by the malware writers' imagination.
- **Backdoor/bot**: Allows an attacker to bypass a security check or grants access to certain resources that were otherwise unreachable, such as a commandline shell.
- **Virus**: Malware that, when executed, tries to replicate itself.

- **Worm**: Similar to a virus but is standalone and spread from machine to machine across networks.
- **Rabbit**: Similar to a worm, but deletes the original copy of itself after replication.
- **Spyware**: Collects information from a computer and transmits it to someone else. Information can be usernames and passwords, e-mail addresses or creditcard numbers.
- **Adware**: Similar to spyware but more focused on the user and their habits.
- **Rootkit**: Is able to hide itself or certain files from the user. This can be done at various levels, such as interfering with function calls in user-mode, or nesting itself inside of the kernel.

A piece of malware can also be a combination of any of the above types of malware. For example malware that is used to create a botnet of zombies includes both the worm and backdoor features in order to both replicate itself over the network while in the meanwhile provide access to the infected host to the administrator of the botnet.

2.2 Static vs dynamic analysis

Program binaries can be analyzed in roughly two ways: using static analysis or dynamic analysis.
In case of static analysis, the program is not being executed, but instead using a for example a `strings`[1] tool, all readable text ASCII and Unicode from the binary is outputted which usually provides good initial clues about the binary. Another way of static analysis is by using a disassembler by which the binary data is interpreted and displayed as a sequence of assembly instructions. A limitation of static analysis however is that when a program has for example been packed by an executable packer it cannot be analyzed, as will be discussed in section 4.4.
In contrast to static analysis, with dynamic analysis the program is actually executed while monitoring the program's behavior. For example, any file system access or network communication is recorded based on which possible malicious behavior can be detected. This paper will primarily focus on the dynamic analysis of programs.

3 Data collection and analysis

This paper focuses on the use of dynamic analysis to collect data about the progra. The term dynamic analysis is used when a the actions of a program are monitored while it is being executed[6]. This monitoring can be performed using various techniques which will be described in the following sections.

3.1 Function call monitoring

In order for a program to either reuse code or to perform higher-privileged operations, Application Programming Interfaces (APIs) have to be used. TheseAPIs are respectively external libraries or system calls which can be accessed by any program. Examples of libraries in the

[1]Sysinternals: Strings - `https://technet.microsoft.com/sysinternals/strings.aspx`

Windows OS are the C Run-Time library[2] which offers various routines like data conversion and -alignment and error handling, and the Windows Sockets 2 library[3] which allows programmers to build network-capable applications.
A special type of API are the system calls. While in most programs the code runs in userspace, some features like writing to a file on the filesystem requires privileged access. Windows exposes these functions via the Windows API (`kernel32.dll`) which redirects the function calls to the Windows Native API (`ntdll.dll`) where the actual system call is executed[10]. The difference between the Windows API and Native API is that the latter one can differ for every Windows version and therefore directly calling functions in the Native API is not supported. In contrast to the Windows Native API, the Windows API remains stable for any given version of the Windows OS and is documented.
By hooking API calls, insight can be gained in the behaviour of the program. A preferred way to get detailed information about calls to the Windows API is by hooking the Native API. An additional advantage of hooking the Native API is that any malware circumventing the Windows API by directly making calls to the Native API will also be captured.

3.2 Approaches for hooking

In order to get hold of the execution flow to understand what the code is doing, code needs to be inserted, for example at places where a function is being called.
In case of availability of the source, this additional code can simply be inserted in the source code, or the compiler or interpreter can be used to insert hooks in the code. In case the source code is not available, there are several approaches to insert hooks.
The first way to insert a hook in a binary is to set a software breakpoint (assembly instruction `int 3`; hex `0xCC`) at the function call or the first instruction of the called function. This will send a signal to the OS and break execution of the program. At this moment all information about the current state of the process is available and can be stored or analysed. The same result can be accomplished by setting a hardware breakpoint or by setting the trap flag in the x86 EFLAGS register[8].
Alternatives to setting breakpoints are replacing libraries and binary rewriting. Libraries that are loaded by the application can be replaced by a proxy library which on its turn can perform the call to the original function. Binary rewriting can be performed in two ways. The first option is updating the binary file on disk, inserting instructions that perform the hook. A second option is to load and then patch the program in memory before starting execution of the code.
Detours[4] is an example of software that can perform both in-memory patching as well as patching the binary. In order to hook a function, Detours inserts an unconditional jump (`jmp imm`) at the first instruction of the function being called, preserving the instructions being overwritten. This jump points to the detour function at which one can do any required interception preprocessing. After executing this custom code, the code that has been overwritten by the jump instruction, is being executed in the so-called trampoline function. After that an unconditional jump is performed back to the instruction right after jump where code instruction pointer originally came from [7]. Once the hook has been inserted, this can be used to obtain all bookkeeping information such as arguments passed to a certain function which

[2]MSDN: Run-Time Routines by Category - `https://msdn.microsoft.com/library/2aza74he.aspx`
[3]MSDN: Windows Sockets 2 - `https://msdn.microsoft.com/library/windows/desktop/ms740673.aspx`
[4]Microsoft Research: Detours - `http://research.microsoft.com/projects/detours/`

provides the investigator insight into the workings of the application. This can be performed automatically, recording a sequence of functions that have been invoked. Optionally, functions can also be grouped like for example the call to `connect` a socket and the subsequent call to `send` or `receive` data with that socket.

3.3 Function parameter analysis

In order to dig deeper into the origin of a certain input parameter recorded from a function hook to learn more about the program's behaviour, information flow tracking can be used. Using information flow tracking, a piece of data can be selected by a *taint source* which means it is interesting data to be analysed. This tainted value will then be followed throughout data manipulations and finally end up in a *taint sink*, a component of the system which is configured to respond whenever a tainted value is passing.
Possible ways in which data can be manipulated throughout a program and require propagation of the tained values are arithmetic operations in registers (i.e. `mov eax,x; sub eax,0xB4`), references via memory addresses (i.e. `mov eax,[x+0x1C]`), and control flow dependencies which lead to an implicit information flow (i.e. `cmp x,0x00; jne false; mov eax,0x00; false:mov eax,0x01`).

4 Implementation

In order to perform the actual analysis of the malware, the privilege level, and related to that, the level of detail available at that level have to be taken into account. Moreover, in case the malware requires network connection, something which nowadays malware frequently needs, a limited Internet connection should be provided. This chapter will first examine the possible execution environments in sections 4.1 and 4.2. After that, the possible ways to provide Internet access to a piece of malware will be discussed in section 4.3. Finally, the categories of anti-monitoring protections will be listed in section 4.4.

4.1 System architecture

Important is that the analysis has to be performed with higher privileges than the privileges at which the malware is running [9]. In case the malware has an equal or even higher level, it is possible for the malware to hide from, or influence the monitoring components which would provide an incomplete or biased view on the malware.
The x86 system architecture consists of multiple protection rings that are listed in Table 1. Ring 0 to 3 are within the virtual machine and managed by the operating system. The Hypervisor at ring -1 manages access to hardware resources while rings -2 and -3 are respectively firmware and the logic inside chips [11].
This means that for example when a rootkit which in case of Windows runs inside of the OS's kernel (ring 0), the malware needs to be monitored from privilege ring -1 or lower.
With the difference in level at which the monitoring takes place versus the level at which the malware is running, a level of abstraction is introduced, which leads to loss of information, also known as the semantic gap [6]. However, to decrease the semantic gap, it is possible to run an additional component within the higher level ring to complement information missing at the lower ring [3].

#	**Purpose**
3	Applications
2	Drivers
1	Drivers
0	Kernel
-1	Hypervisor
-2	System Management Mode (SMM)
-3	Chipset

Table 1: Privilege Rings

4.2 System setup

Depending on the type of analysis as described above, a specific type of setup can be chosen to analyse the malware. This section will explain each of these setups and explain the advantages and disadvantages.

User/Kernel space Whenever malware is running in user space (ring 3), it can be monitored from kernel space (level 0) without noticing the monitor's presence. From the kernel, the set of APIs used by the userspace program can be resolved directly for improved analysis.

Emulator In contrast to running malware directly in an operating system which runs directly on the hardware, an emulator provides an emulated CPU and memory that are abstracted from the actual hardware CPU and memory. During execution, the instructions are read from the binary and equivalent instructions are being executed in the emulated environment. Any libraries used by the malware need to be implemented by the emulator in order to successfully run the malware. This gives one full control over the malware being executed. An example of such emulator is Wine[5] [12].

Virtual machine An alternative to an emulator is a Virtual Machine (VM) which abstracts the hardware layer and using Virtual Machine Monitor (VMM), a also known as hypervisor, presents the virtual hardware to the programs running in the VM. Non-privileged instructions are run directly on the (physical) hardware while the VMM emulates any privileged instructions. Moreover, most virtualization products also provide the possibility to create snapshots and revert to snapshots, which comes in handy for a researcher to for example quickly revert back to a moment at which the malware has not installed itself yet.

4.3 Network connectivity setup

In case the malware requires network or Internet connectivity, there are two ways to set this up.

Virtual network The first way is to provide a fake network which makes all DNS requests resolve to a local IP address and emulates different local services such as an HTTP, FTP or IRC server locally, storing all of the logs. A tool which is able to perform such emulation in

[5]Wine website - `https://www.winehq.org/`

Windows is FakeNet[6]. This is a safe way to get malware working in a contained environment, however a disadvantage is that some malware expects binaries (i.e. in case of a dropper) or commands (i.e. botnet) from a server on the Internet.

Filtered Internet access In order to make the environment as realistic as possible and get most results, access to the Internet should be allowed. However, because the malware will probably perform malicious activities, the access has to be restricted and closely monitored. The malware might for example receive commands to perform a Denial of Service (DoS) attack on a certain IP address or start scanning an IP range in order to exploit vulnerable services. These activities can be limited to a minimum by applying traffic shaping and rate limiting.

4.4 Complicating factors

As malware writers try to keep their behaviour secretive to prevent researchers from developing effective countermeasures, they utilize various methods to conceal the working of the malware. The following paragraphs elaborate more on the methods used.

Packers and self-modifying code In order to complicate both static and dynamic analysis, packers are applied to a binary for obfuscation. An example of a popular packer is UPX. UPX takes all sections from a binary and packs them into a single section. It then adds the unpacker code in the new executable which on execution takes the packed section, expands it in memory, in case the imports table has been packed too, resolves all the imports from the original executable and then jumps to the Original Entry Point (OEP) [10]. Additionally, some packers perform multilayer-packing or use methods described in the following paragraphs.

Detection of analysis environments To avoid monitoring by researchers, malware often makes use of anti-virtualization and anti-debugging techniques. The implementations of these anti-monitoring measures can be categorized in the following four areas [5].

- ***Hardware:*** In case of virtualization, as the physical hardware is virtualized, virtual hardware is offered to the VM. The device IDs and names can be easily enumerated by the malware and matched against a list of virtual hardware components. Moreover, the driver used to improve performance of the virtualised OS can be recognized.

- ***Execution environment:*** Products like VMware and VirtualBox have a communication channel between the host and the VM. This channel can be detected, as well as presence of the VMware Interrupt Description Table (IDT). Moreover, when executing an application within a debugger, via its API Windows provides functions like `IsDebuggerPresent()`[7] and `CheckRemoteDebuggerPresent()`[8] which make it possible for a process to check whether it is being debugged. Lastly, debugger leave more artefacts such as when setting software breakpoints. As described in section 3.2, an

[6]Sourceforge: FakeNet - `https://sourceforge.net/projects/fakenet/`

[7]MSDN: IsDebuggerPresent - `https://msdn.microsoft.com/library/windows/desktop/ms680345.aspx`

[8]MSDN: CheckRemoteDebuggerPresent - `https://msdn.microsoft.com/library/windows/desktop/ms679280.aspx`

interrupt instruction is written to the memory of the process. A process can then for example calculate a checksum over the memory to be executed and detect there have been modifications.

- ***External applications:*** In virtual environments such as VMware, tools like VMware tools are installed for easy interaction between the VM and the host. Additionally, presence of debugging tools like WinDBG, Olly and IDA Pro, both on disk as well as in memory can be easily detected and cause malware to behave differently.
- ***Behavioral:*** When single-stepping through a program, the execution obviously takes much longer. A malicious program can detect this by calculating the amount of milliseconds it took to get from a certain place in the code to another one. Two examples of how to perform this check are the `rdtsc` assembly instruction and `QueryPerformanceCounter()`[9] or `GetTickCount()`[10] functions from the Windows API [10].

Depending on which type of setup of the system is used, these methods can be detected by the program being monitored [12].

Logic bombs Instead of simply executing its malicious payload, malware can also include conditions limiting the moments at which it will run its malicious payload. Examples of such conditions is the payload only running at a certain date or only after a certain number of keys have been pressed by the user [6].

5 Network data capturing and reverse engineering

Automatic protocol reverse engineering aims to retrieve the protocol format and the protocol state machine. The protocol format is the structure of the message beings sent. The protocol state machine details in which sequence the messages are valid.

Deducing the protocol format usually consists of two steps: First, using a set of protocol messages, extract the message format of each message. Then, from this set of message formats, deduce which fields are: repetitive, optional or alternative. From this, the protocol format can be inferred.

Caballero et al. [4] suggests capturing the message format in a message field tree. In this tree, each node represents a field. The root node represents the entire message. A child node represents a subfield of it's parent. As such, the leaves together represent the entire message, each containing the smallest subfield. Each represents the formatting information of the field. This includes the field length and whether it's variable and the dependencies between fields—e.g. the length field depends on all other fields.

In addition to constructing the message format, it is very important what each field represents. The semantics are represented in the message field tree as the labelling of each node.

Caballero et al. [4] focusses on extracting the message field tree of the messages that are sent by the application as well as inferring the semantics of each field for received and sent messages. In order to accomplish this, they deduce the information from the way the

[9]MSDN: QueryPerformanceCounter - `https://msdn.microsoft.com/library/windows/desktop/ms644904.aspx`

[10]MSDN: GetTickCount - `https://msdn.microsoft.com/library/windows/desktop/ms724408.aspx`

output buffer is constructed. The output buffer is constructed from other buffers containing the information included in each field.

5.1 Capturing

First a trace is constructed using dynamic taint tracking while the network functions of the program are used. This trace contains all instruction level executions, the content of the operands and the associated taint information. In this trace each message is considered separately. Splitting the trace into individual messages is done by splitting it whenever data is written to a socket. Except if the argument detailing how many bytes to be read is tainted, in this case the data is still considered part of the previous message. This is to prevent a read entailing the length of the payload to be split from the payload itself.

This approach is resilient to code obfuscation aimed at preventative static analysis. However, this approach is sensitive to methods that prevent dynamic analysis as entailed in 4.4 [4].

5.2 Field attribute inference

In order to deduct the message field tree from the trace several steps are taken. First the data is prepared so the buffer deconstruction can be done efficiently. After the buffer deconstruction, the message field tree is constructed.

5.2.1 Preparation

In the first phase, the trace is analysed to collect information required by the later phases and the semantics inference. It checks at which point a loop is being executed in the trace. It also does a callstack analysis, indexing the functions by instruction number. So that the innermost function containing an instruction is known by the instruction number. Lastly, it analyses at which point in the execution a memory location contains one buffer and at which point it contains a different buffer.

5.2.2 Buffer deconstruction

In the buffer deconstruction, a buffer is taken and deconstructed into different buffers of which it comprises. These buffers are then deconstructed in turn until no new buffers are found. A dependency chain is built, deriving for each byte in the initial buffer where it originated. If a memory location is found, a new dependency chain in made for that buffer. If a constant value is found— e.g. an immediate value or an XOR operation of a registry on itself the dependency chain is ended. If an operation on the location is found, it is assumed that this is a leaf field as operations on nodes with children do not occur.

5.2.3 Extracting the buffer structure

Once we have a dependency chain for every byte in the initial buffer, we can construct the message field tree. The last element in each dependency chain is the source. For a buffer, two elements that are in memory are part of the same field if they are in adjacent memory locations and neither location has been overwritten in between their write operations. If they are not in memory, they are part of the same field if they were written by the same instruction. By iterating up from the initial fields (the leaves in the tree) the rest of the tree is constructed using the same method.

5.3 Semantics inference

For fields of the sent or received message, the program knows their semantic meaning. In many cases the program will have used fields of the sent message and will use fields of the received message. When these fields are used on a known function—e.g. using the ip to connect through a socket—we can derive from that the semantics of that field.

5.4 Dealing with encryption

In order to be able to analyse messages that are sent and received encrypted Caballero et al. [4] uses a heuristic. The number of arithmetic and bitwise operations in a function divided by the total number of instructions of the function. If this value exceeds a certain threshold, the function is considered an encrypting or decrypting function. For outgoing messages, the read set of the encryption function is deduced. For incoming messages the write set of the decrypting function is deduced. From these sets the unencrypted buffers are selected as those that vary in content between multiple function calls.

6 Automatic malware classification

Malware classification is dividing malware up according to their behaviour. This is an interesting field of research because currently anti virus products struggle with polymorphic malware. If they could analyse a file and determine it behaves like known malware that would be a large improvement.

6.1 Behaviour

Wagener et al. [12] define the behaviour of a piece of malware as the sequence of system calls it performs. [12] On the other hand Bailey et al. say the following: "We define the behavior of malware in terms of non-transient state changes that the malware causes on the system" [2]. They give spawned process names and network connection attempts as examples of such state changes.

6.2 Distance

The aim is to group malware with similar behaviour together. In order to do this we need a distance metric to determine whether behaviours are similar. The first candidate that comes to mind is the edit distance[12, 2]. As the distance between the behaviours is smaller, they are more similar. However, similar malware does not always make calls or state changes in the same order. It might be that the order does not matter and they perform them in the opposite order.

In Wagener et al. [12] the Hellinger distance is used instead. This distance compares the frequency with which certain calls are made by certain malware. If the frequency is similar, the behaviour is also similar. We note that this loses all ordering information.

Bailey et al. takes a different approach using the normalized compression distance (NCD).

$$NCD_z(x,y) = \frac{Z(xy) - min(Z(x), Z(y))}{max(Z(x), Z(y))}$$

They take xy to be the concatenation of two behaviours x and y. Z is the compression function, here zlib is used to compress the representations of the behaviour. The more similar x and y are, the smaller the compression of their concatenation is. As such a lower NCD means x an y are more similar. We note that some ordering information is retained as identical sequences lead to a smaller compressed size. However, this depends on the degree to which the compression function manages to compress the information contained in the behaviours.

6.3 Phylogenetic tree

A phylogenetic tree shows the evolutionary change in the exposed behavioural profile. The tree is build up by taking each malware behaviour as a node. Then, the two nodes with the smallest distance are connected through a parent containing their distance. This process is repeated until all nodes are part of the tree. When comparing to a group, distance to the closest node is chosen. In other words, the distance between two groups is the distance between the closest behaviours. [12]

References

[1] Aycock, J. *Computer viruses and malware.* Springer Publishing Company, Incorporated, 2010.

[2] Bailey, M., Oberheide, J., Andersen, J., Mao, Z. M., Jahanian, F., and Nazario, J. *Recent Advances in Intrusion Detection: 10th International Symposium, RAID 2007, Gold Goast, Australia, September 5-7, 2007. Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, ch. Automated Classification and Analysis of Internet Malware, pp. 178–197.

[3] Bayer, U., Moser, A., Kruegel, C., and Kirda, E. Dynamic analysis of malicious code. *Journal in Computer Virology 2*, 1 (2006), 67–77.

[4] Caballero, J., Poosankam, P., Kreibich, C., and Song, D. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 621–634.

[5] Chen, X., Andersen, J., Mao, Z. M., Bailey, M., and Nazario, J. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on* (2008), IEEE, pp. 177–186.

[6] Egele, M., Scholte, T., Kirda, E., and Kruegel, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv. 44*, 2 (Mar. 2008), 6:1–6:42.

[7] Hunt, G., and Brubacher, D. Detours: Binary interception of win32 functions. In *Third USENIX Windows NT Symposium* (July 1999), USENIX, p. 8.

[8] Intel, R. and ia-32 architectures software developer's manual volume 1: Basic architecture. *Intel Corporation*, 77.

[9] Rossow, C., Dietrich, C. J., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., Bos, H., and v. Steen, M. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP), 2012 IEEE Symposium on* (May 2012), pp. 65–79.

[10] Sikorski, M., and Honig, A. *Practical malware analysis: the hands-on guide to dissecting malicious software.* no starch press, 2012, pp. 17,358,384–402.

[11] Szefer, J., and Lee, R. Architectural support for hypervisor-secure virtualization.

[12] Wagener, G., State, R., and Dulaunoy, A. Malware behaviour analysis. *Journal in Computer Virology 4*, 4 (2007), 279–287.