Building Malware Infection Trees

Jose Andre Morales[1], Michael Main[2], Weiliang Luo[3], Shouhuai Xu[2,3] and Ravi Sandhu[2,3]
[1] Software Engineering Institute, Carnegie Mellon University*
[2] Institute for Cyber Security, University of Texas at San Antonio
[3] Department of Computer Science, University of Texas at San Antonio

Abstract

Dynamic analysis of malware is an ever evolving and challenging task. A malware infection tree (MiT) can assist in analysis by identifying processes and files related to a specific malware sample. In this paper we propose an abstract approach to building a comprehensive MiT based on rules describing execution events essential to malware infection strategies of files and processes. The MiT is built using strong and weak bonds between processes and files which are based on transitivity of information and creator/created relationships. The abstract approach facilitates usage on any operating system platform. We implement the rules on the Windows Vista operating system using a custom built tool named MiTCoN which was used in a small scale analysis and infection tree creation of a diverse set of 5800 known malware samples. Results analysis revealed a significant occurrent of our rules within a very short span of time. We demonstrate our rule set can effectively and efficiently build infection trees linking all related processes and files of a specific malware sample with no false positives. We also tested the possible usability of a MiT in disinfecting a system which yielded a 100% success rate.

1 Introduction

The release of never before seen malware into the wild poses a severe global threat to vulnerable systems given the difficulty to detect via signature based anti-malware programs. Possible detection can be achieved with heuristics but not guaranteed to fully eradicate the malware which leaves disinfection as the next best option. To comprehensively disinfect a system, the malware must be analyzed. The analysis must effectively and efficiently detail the infection process in a meaningful way primarily documenting the files and processes which are created or modified by the malware. Analyzing the infection process can be aided by building a malware infection tree (MiT). A MiT is a directed tree structure where each node represents a file or process and each edge represents the execution event rule causing node to join the tree. To correctly build a MiT, an understanding of the essential characteristics of malware infection is required. From the seminal definitions provided by Cohen [2] and Adleman [1], an executable file labeled a virus has the fundamental ability to self-replicate which we consider to be a basic construct for a MiT and it is essential to understand the different ways in which a virus can implement this construct on various operating systems. Previous work such as [10, 9] has shown some implementations of self-replication while others have attempted to record malware behavior using various graph structures [7, 8]. A more comprehensive MiT includes processes modified by malware. There are many known ways [12, 3] in which a process modifies other already running processes. This technique is primarily implemented via a memory injection resulting in the modified process performing anomalous, and often nefarious, events. Self-replication and memory injection create a strong bond between related processes and files and are the basic constructs of our MiT. Our MiT is further enhanced with constructs recoding the creation of files and processes by a malware not involving self-replication or memory injetion which create weak bonds. Implementing the various ways in which a malware infection can occur is highly OS dependent. It is imperative to collect needed data of a malware infection in as low a level as possible to assure building of a comprehensive MiT. Some malware, such as a rootkit [5, 13] will execute at deep or privileged OS levels hiding and avoiding detection while infecting the system. In this paper, we present an abstract approach to building MiTs using execution events rules. The rules describe execution events essential to malware infection strategies

*This research was performed in the University of Texas at San Antonio

on files and processes. MiTs are built based on strong and weak bonds between relevant files and processes. We describe rule implementation in the Windows Vista operating system with our custom tool, named MiT-CoN, that builds MiTs in real time. The tool analyzed and built MiTs of over 5800 diverse known malware samples. We evaluate the efficiency of our approach by recording system stability during MiT creation and timewise analysis of MiT creation. Effectiveness was evaluated by measuring frequency of rules, timewise occurrence of rules, a comparison to infection structures of comparable systems for false positive production, and attempt system disinfection using only the MiT as a guide. Our analysis revealed our MiTs were constructed within 7 seconds of initial execution without any noticeable system instability . Our disinfection attempts yielded a 100% success rate implying MiTs may be useable in real system disinfection scenarios. The contributions of our paper are as follows:

- Propose an abstract approach to building malware infection trees (MiTs).
- Define execution event rules describing essential components of infection strategies.
- Describe implementation in the Windows Vista OS User and Kernel levels.

Several comparable systems such as Anubis, BitBlaze, JoeBox, CwSandBox, and Malheur perform dynamic analysis of a submitted sample and return a tree like structure of related files and processes. The tree edges are based on operating system specific execution events which objectively link two nodes together. The critical problem is tree creation is based on interpretation of collected execution data which can result in a tree that loosely relates files and processes which may not even be part of the malware infection such as processes and files routinely used in standard OS operations. This misinformation does not correctly represent a malware infection and can result in false positives. Our approach creates more meaningful trees by creating execution event rules based on fundamental malware infection characteristics producing fundamental bonds between nodes possibly reducing false positives. The result of our approach is a tightly bound tree structure containing minimal or no non-related data leading to a minimization of false positives and a much more realistic representation of the analyzed malware's infection strategies. The rest of this paper is organized as follows: Section 2 presents the rules used to create a MiT, Section 3 is our tool implementation, Section 4 is our analysis & results, and Section 5 is conclusions & future work.

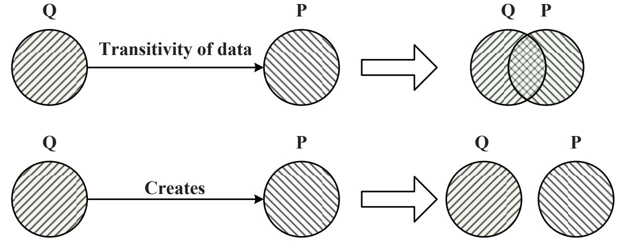

Figure 1. MiT strong & weak bonds

2 MiT Construction Rules

Strong and Weak Bonds. An essential component of our MiT building approach is linking nodes together in a strong or weak bond, illustrated in Figure 1, relationship based on the malware infection related interaction between two nodes during execution. We define a strong bond between a source node Q and destination node P when a transfer of data from Q to P occurs. The data is part of Q and transferring it to P strongly bonds both nodes based on the transitivity of data creating an intersection of identical data between Q and P. We define a weak bond when a source node Q arbitrarily creates a destination node P and there is no transitivity of data. This weak bond can be viewed as a creator/created relationship since P exists because of Q but there is no intersection of identical data between both. Other comparable systems create infection trees based only observed execution events and their nodes are not linked in a fundamentally meaningful way allowing addition of non-related nodes to the tree. The bonds between nodes of our MiTs are defined in the fundamental realm of malware infection strategies producing an infection tree with the reduced likelihood of including non-malware related nodes. A strongly bonded MiT will consist of essential files and processes directly descending from the original malware executable due to the transitivity of data between nodes and should be eradicated first in a disinfection strategy to prevent further infection and injury. Enhancing the MiT with weak bonds provides those files and processes that may not be essential to the malware's infection and injury but still should be eradicated during disinfection.

Construction Rules. We define a malware infection tree (MiT) as the output of a function $M(\cdot)$ on inputs X with a set of execution event rules R, where X is the executable file being analyzed, and R contains a set of rules $r_1 ... r_n$ that define the conditions under which an object becomes part of a MiT. $M(X)$ outputs a MiT with a directed graph as MiT $= (N, E)$, where

N is a set of nodes $n_1 ... n_z$ with $z \geq 2$, and E is a set of ordered pairs of nodes $(n_i, n_j) \in N^2$ that create an edge between n_i and n_j. A node n is either a file in the file system or a currently running process. We assume X is always the root node of a MiT. Once a new node n is added to a MiT, its infection execution events are recorded and used to add further nodes to the MiT. If $z = 1$, then only one node n, presumably the root node X, is present in the MiT and thus no tree was created. MiT construction is based on the rules in R. The rules are mostly based on the fundamental definition of malware, Cohen [2] & Adleman [1], which stipulate a malware, specifically viruses and worms, must replicate to be classified as such. The rules also reflect a malware's tendency to nefariously modify running processes.

File System Rules. We consider a malware can infect via self replication into the file system in two primary ways. First, using a call such as $copy(n, m)$, where n is a node of a MiT and the caller of the *copy* function. In this case n creates a new file which is an exact copy of itself. Second, using a series of calls such as $read(n, q); write(q, m)$, where n is a node of a MiT and the caller of $read$ and $write$, q is some temporary storage, and m is an already existing file that is being modified by n. The modification can be achieved by n prefixing, suffixing, overwriting or randomly writing some or all of its own data into the file m. Note in both cases, n is the source of the replications and the caller of the operations, therefore n is invoking self-reference replication as described in [9]. We also consider a malware that can infect a system without requiring self replication. This can be donw through the arbitrary creation of files such as $createfile(m)$ in the file system, where an existing node n calls the operation and creates a file that contains no data originating in n. There are many known malware samples that create files during execution for several reasons such as logs, configurations or to store data later sent to a remote host. Even though these files are not created via self replication they are valid components of an infection and should be included in a MiT. Based on these considerations, we define the following three file system execution event rules for the construction of a MiT:

$f1$**:Infection via self replication.** A file m becomes a new node $n \in N$ of a MiT if and only if $P(replicate(P, m)) \rightarrow True$ where a currently running process P is a pre-existent node in a MiT and P has issued a replication request where P itself is the source parameter and m is the destination parameter. m becomes a new node in the MiT where the node for P is located and the outgoing edge (P, m) is labeled $f1$. Note that m can be a newly created file by P or an existing file and the replication from P to m can be complete or partial. $f1$ exhibits transitivity of data and is labeled a strong bond.

$f2$**:Infection via arbitrary file creation.** A file m becomes a new node $n \in N$ of a MiT if and only if $P(filecreate(m)) \rightarrow True$ where a currently running process P is a pre-existent node in a MiT and P has issued a file creation request where m, a newly created file, is the destination parameter. m becomes a new node in the MiT where the node for P is located and the outgoing edge (P, m) is labeled $f2$. $f2$ is a creator/created relationship with no transitivity of data and is labeled a weak bond.

$f3$**:Infection via arbitrary file write modification.** A file m becomes a new node $n \in N$ of a MiT if and only if $P(filewrite(m)) \rightarrow True$ where a currently running process P is a pre-existent node in a MiT and P has issued a file write request where m, a pre-existing file, is the destination parameter. m becomes a new node in the MiT where the node for P is located and the outgoing edge (P, m) is labeled $f3$. Note P may modify m either by prefixing, suffixing, overwriting or randomly writing data into m. The essential component of $f3$ is the data being written to m comes from some other location and not from P. $f3$ does not exhibit transitivity of data and is labeled a weak bond.

Process Rules. We consider malware that can infect a system via process manipulation in two primary ways. First, a malware can modify the memory range of a currently running process. This is typically done via dynamic code injection [12, 3]. A malware will write (inject) nefarious code into the allocated memory of some other process and then spawn a new process instance which executes the just injected code. This results in the victim process performing execution events it would otherwise not do under benign conditions. Several known malware inject nefarious code into system critical processes. Malware authors assume there is a high unlikelihood that a user would terminate these processes since they are considered critical to OS functionality. Second, malware will create processes from the static file images of executables that either were created or downloaded to the system. Malware is known to copy or download from remote malicious servers to the system other malware as part of its payload, in many cases a trojan(s) such as password stealer, key loggers, and spam engines. In some cases, a malware may self replicate by spawning multiple processes of itself running on a system to either overwhelm the system or survive an anti-malware detection and removal attempt. Process manipulation is a very powerful tool for malware facilitating: system compromise, increased chances of detection survival, and delegate

nefarious goals to benign processes. Injection is particularly powerful allowing nefarious deeds to possibly go unnoticed when carried out by benign system critical processes. Based on these considerations, we define the following two process execution event rules for the construction of a MiT:

$p1$:Infection via dynamic code injection of a currently running process. The static file image m of a currently running process P becomes a new node $n \in N$ of a MiT if and only if $Q(codeinject(P, d)); Q(newprocessinstance(P)) \rightarrow True$ where Q is some currently running process and a pre-existent node of a MiT, writes data, presumably code instructions stored in Q's memory space or static file image, into the allocated memory space of P and then spawns a new instance of P which executes the just written data. m becomes a new node in the MiT where the node for Q is located and the outgoing edge (Q, m) is labeled $p1$. P is a pre-existing and presumed benign currently running process of the system which gets nefariously modified by Q in memory. The static file image m of the modified process P is never modified by Q. $p1$ exhibits transitivity of data and is labeled a strong bond.

$p2$:Infection via process spawning. The static file image m of a currently running process P becomes a new node $n \in N$ of a MiT if and only if $Q(newprocess(m)) \rightarrow True$ where Q, a currently running process and a pre-existent member of a MiT, invokes the command to create a new process P from a static file image m. The static file image m is also a pre-existent node of the same MiT as Q. An outgoing edge (Q, m) is created and labeled $p2$ which stores information about the static file image m and the newly created process P. In this scenario, Q spawns new processes from files that are already part of the malware's MiT including Q's own static file image. Some of these files were either created or downloaded by some node of the MiT. The key element in this rule is that m is a malware related file and part of the MiT. As opposed to $p1$ where malware injects nefarious code into benign processes then creates a new instance, $p2$ creates a process from a nefarious static file image m. The nefariousness of m is based on its pre-existing inclusion as a node of a MiT. It is possible for m to be created by a process R using the file system rules above and then spawned as a new process by Q. Both R and Q are separate nodes of the same MiT, this would produce two incoming edges to m: one for the file creation and one for the process spawning. Note that m can be the static file image of Q, meaning $p2$ also records when Q creates a new process instance of itself. In this case the outgoing edge (Q, m) contains the same static file image information for m and different identification information for the newly spawned process P. ps is a creator/created relationship with no transitivity of data and is labeled a weak bond.

Implementing the file system and process rules can produce an intersection of usage based on the target OS. For example, $f2$ and $f3$ may be invoked as subroutines of $f1$, $f3$ may be invoked as a subroutine of $f2$, and $f2$ can be invoked as a subroutine of both $f1$ and $f3$. Another example is the OS performing code injection to create both new processes and instances of already running processes. In this case, extra information about the process and its injector must be acquired to adequately decide if $p1$ or $p2$ has occurred. To allow some reasonable flexibility of the rules to accommodate unavoidable intersections we assume the execution event rules describe abstract scenarios that may invoke other rules as subroutines and explicitly recognizing the use of these other rules is not required. A MiT can have multiple edges resulting from file system and process rules occurring on the same node. When building a MiT, it is allowable to have multiple edges between nodes reflective the rule justifying the edge's existence. A possible scenario may be a process P creates a file m which then creates a file o. The file o is then spawned as a new process by P. This would create two incoming edges to o, the first edge is (m, o) labeled $f1$ and the second edge is (P, o) labeled $p2$. Having a multi-edged MiT provides richer data for disinfection. Based on the MiT a user can assess which nodes should be dealt with first, perhaps using the number of incoming and outgoing edges as a weighted determination scale. Those nodes with greater number of incoming/outgoing edges may receive higher priority over other less populated nodes. Once a MiT is created, some of the files and processes belonging to the MiT may no longer exist. Several known malware samples delete files and terminate processes which may belong to its own MiT. A typical case may be a malware X creates several descendant files some of which are spawned as new processes. Then some time later one of these descendant processes terminates X and/or erases its static file image from the file system. Another scenario is X or any other spawned processes terminating itself and/or deleting its own file from the system. These scenarios do not invalidate the MiT as the file or process did in fact exist at the moment of addition as a node to the MiT. The key challenge in implementing these rules is understanding the various ways a file or process can be created or modified in a specific OS. Once this is understood, an implementation can be created at an appropriate OS level that captures all or most of the studied implementations. We address this challenge in

the next section which presents our MiT construction tool for the Windows Vista OS platform.

3 MiTCoN: Windows Vista Implementation

Our malware infection tree construction tool, named MiTCoN, is a Windows command line application which implements the execution event rules in the Windows Vista platform which outputs a table representation of a MiT. The MiT is built in real time by monitoring the samples' execution behaviors. MiTCoN takes as input the absolute path of the target WIN32 PE executable set as the MiT's root node which facilitates MiT building by knowing the process from which to start monitoring execution behaviors.

Implementing File System Rules. To detect when $f1$, $f2$, and $f3$ occur by some process P, MiTCoN traces a set of file system functions located in the Windows kernel using a form of function hooking [5, 13]. These functions belong to the Zw family [14] and are located in the Windows SSDT table [11]. Windows provides several ways to create and modify files at the user level. When these commands get passed down to the kernel level, Windows merges them into a handful of Zw functions. MitCon traces two sequences of Zw function calls that successfully implement $f1$, and one sequence for $f2$ and $f3$. The first sequence to determine the occurrence of $f1$, infection via self replication, for some process P, MiTCoN traces the sequence of Zw functions with appropriate parameters listed in Table 1. The key to determining that $f1$ has occurred, is to establish that P is referencing itself and is the source of the write operation. According to Table 1, self-replication starts with P opening itself with read access in ZwCreateFile where sourcepath is the absolute path of P. The functions ZwCreateSection and ZwMapViewofSection use the file handle returned from ZwCreateFile to map the data that is going to be written from P into memory. Finally, ZwWriteFile reads the data stored in baseaddress, which is returned by ZwMapViewofSection, into the target file. At this point, an instance of $f1$ has been completed by P. Note in MapViewofSection, the in parameter processhandle is assured to be the value -1. This indicates the mapping will be of the caller process P. This sequence of calls is used when P makes an exact copy of itself in a newly created file.

The second sequence to determine the occurrence of $f1$ for some process P, MiTCoN traces the sequence of Zw functions with appropriate parameters listed in Table 2. The second sequence in determining $f1$ is simpler involving only two Zw functions. The source path in ZwReadFile refers to the absolute path of P and the memaddress is used as temporary storage of the data from P which is written to targetfilepath in ZwCreateFile. This sequence of calls is used primarily when P is self replicating into already existing files.

```
ZwCreateFile(in:read_access, in:sourcepath, out:filehandle);
ZwCreateSection(in:filehandle, out:sectionhandle);
ZwMapViewofSection(in:sectionhandle,
   in:processhandle, out:baseaddress);
ZwWriteFile(in:baseaddress, out:targetfilepath)
```

Table 1. 1st Function Sequence Used in $f1$

```
ZwReadFile(in:sourcepath, out:memaddress);
ZwWriteFile(in:memaddress, out:targetfilepath);
```

Table 2. 2nd Function Sequence Used in $f1$

To determine the occurrence of $f2$, infection via arbitrary file creation, for some process P, MiTCoN traces with appropriate parameters the ZwCreateFile function as listed in Table 1. The only parameter considered is sourcepath which is assured not to be the name and file system location of the caller process P. With this assurance, P, the caller process, is creating a completely new file with the name and file system location stored in sourcepath. To determine the occurrence of $f3$, infection via arbitrary file write modification, for some process P, MiTCoN traces with appropriate parameters the sequence of functions listed in Table 2, which is one of the sequences used for $f1$. The difference is in establishing an occurrence of $f3$, the sourcepath parameter in ZwReadFile is assured not to be the file system location of P. With this assurance $f3$ is determined since the data being written is from some other part of the system and not from P.

Implementing Process Rules. To detect when $p1$ and $p2$ occurs by some process P, MiTCoN performs function hooking on kernel level Zw functions and user level API functions. Several of the various ways in which a process can be injected and spawned at the user level filter down to a handful of Zw functions in the kernel. In Windows, a process can be spawned in two primary forms: a WIN32 process and a Windows service. MiTCoN traces one sequence of function calls for $p1$, and two sequences for $p2$. To determine the occurrence of $p1$, infection via dynamic code injection of a currently running process, for some process P, MiTCoN traces with appropriate parameters the sequence of function calls listed in Table 3. This sequence of functions facilitates the injection of data between process allocated memories. MiTCoN assures P is the

caller of all three functions. Memory is first allocated with ZwAllocateVirtualMemory in the process identified by processhandle starting at baseaddress. these two parameters are again used to write (inject) data by P in the allocated memory of processhandle. Finally P creates a new instance of the just injected process with CreateThread which causes the new instance to execute the newly written nefarious code.

```
ZwAllocateVirtualMemory(in:processhandle, out:baseaddress);
ZwWriteVirtualMemory(in:processhandle, in:baseaddress);
ZwCreateThread(in:processhandle, out:threadhandle);
```

Table 3. Function Sequence Used in $p1$

The first sequence to determine the occurrence of $p2$, infection via process spawning, for some process P, MiTCoN traces with appropriate parameters the function call listed in Table 4. This single function suffices to create a new process from the static file image detailed in objectattributes. The function returns a handle, in processhandle, to the newly created process. MiTCoN checks if the newly created process is of a static file image that is already a node of P's MiT. If yes, then an edge is added.

```
ZwCreateProcess(in:objectattributes, out:processhandle)
```

Table 4. 1st Function Sequence Used in $p2$

The second sequence to determine the occurrence of $p2$ for some process P, MiTCoN traces with appropriate parameters the sequence of function calls listed in Table 5. CreateService will use the file located in BinaryPath as the service to be registered and returns in servicehandle a handle identifying the service. This handle is used in both OpenService and StartService, the end of which results in the service running on the system. Note the function CreateService has an additional parameter that, given the proper value, can start the service immediately. If this occurs, MiTCoN will not need to detect the invocation of OpenService and StartService.

```
CreateService(in:BinaryPath, out:servicehandle);
OpenService(inout: servicehandle);
StartService(in: servicehandle);
```

Table 5. 2nd Function Sequence Used in $p2$

MiTCoN assures P is the caller process for every function in a sequence by invoking GetCurrentProcess() at both the user and kernel levels. As rules are identified, the appropriate MiT is updated with new nodes and edges. The completed MiT presents all the files and processes identified as the source or destination of a given rule. The number of edges going in and out of any one node represents the number of rule instances.

MiTCoN Example. We present the MiT output of MiTCoN in Table 6 and its tree graph in Figure 2 for the malware Backdoor.Win32.Poison. Table 6 has three columns: Source identifies the caller process of the operation(s), Rule(s) gives the execution event rules invoked by Source, and Destination gives the target of the execution event. Each row is a complete invocation of a rule forming a node pair (source,destination) and Rule is the label for the outgoing edge. The first row's Source file name is the root of the MiT. In the first row, Poison performed rule $f1$ on svchest.exe, read as: Poison self-replicated into svchest.exe. The second to last row shows svchest.exe performed rules $f1$ and $p2$ on svchvst.exe, read as svchest.exe replicated into and spawned new process svchvst.exe. Poison has four child nodes: svchest.exe. 1.bat, 1.reg, and 1.vbs. Of these four, Poison self-replicated ($f1$) into svchest.exe and 1.vbs and created ($f2$) 1.bat and 1.reg.

Source	**Rule(s)**	**Destination**
Poison	$f1$	svchest.exe
Poison	$f2$	1.bat
Poison	$f2$	1.reg
Poison	$f1$	1.vbs
Poison	$p1$	wscript.exe
wscript.exe	$p1$	cmd.exe
cmd.exe	$p1$	regedit.exe
cmd.exe	$p1$	attrib.exe
cmd.exe	$p1$	reg.exe
cmd.exe	$p1$	reg.exe
cmd.exe	$p2$	svchest.exe
cmd.exe	$p1$	ping.exe
svchest.exe	$f1$,$p2$	svchvst.exe
svchvst.exe	$f1$	svchist.exe

Table 6. MiT of Backdoor.Win32.Poison

4 Evaluation & Results

Using MiTCoN, we analyzed and built MiTs for 5800 diverse known malware samples randomly selected from GFI Sandbox malware repository [4] uploaded between April and June 2011. According to Kaspersky anti-virus, the samples were classified as: Trojans, Worms, Viruses, Adware, Spyware, FakeAVs, Monitors, Risk Tools, PSW Tools, Hoaxes, web Tool Bars, Porn Dialers, Downloaders, Remote Admin Tools, IRC Clients, P2P worms, Email worms, Backdoors, Bots, Bankers, Clickers, Ransom, Packed, Game Thiefs, Exploits, Rootkits, and Droppers. Analysis was

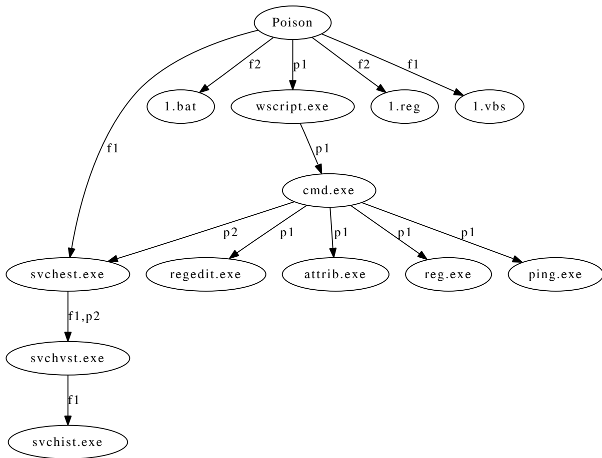

Figure 2. MiT graph of Backdoor.Win32.Poison

conducted in VMWare Workstation with a fresh install of Microsoft Windows Vista logged into the default administrator user account. MiTCoN and each sample were executed for three minutes, the MiT enhanced with timestamps was saved to a database and the snapshot reverted to a clean state. MiTCoN proved to be highly efficient by not causing system instability or crashes during analysis, CPU usage per sample peaked at 3% and averaged less than 1%. Total RAM memory usage never surpassed 14MB with MiTCoN executing. MiTCoN never took more than 7 seconds to build and record a MiT and averaged these operations at 3.1 seconds. There was a high frequency of rule occurrence in all analyzed malware samples with overall totals as follows: $f1$:662, $f2$:14396, $f3$8629, $p1$:647, $p2$:3490 with the top three malware classes: Trojan: 293, Worm:63, Backdoor:56. Every analyzed sampled had multiple occurrences strong bond rules implying strongly bonded MiTs may suffice to understand the malware's essential infection strategy with weak bonds enhancing the MiT. In the instances of $p1$, we infered the malware was delegating nefarious deeds off to seemingly benign processes in order to achieve their goals while not being identified. The majority of processes were spawned as WIN32 with a several being pre-existing nodes in the MiT itself ($p2$). Many files where spawned as a Windows service and not a regular WIN32 process. Detecting services proved critical to building comprehensive trees since the node's subtree was substantial. Many nodes had multiple incoming and/or outgoing edges which illustrates a strong bond between file and process manipulation by malware. The most common combination was $f1, p2$ where malware would self replicate then spawn the process to have multiple instances running on the machine. We conjecture this avoids complete malware eradication or overwhelms the system facilitating compromise. Analyzing timestamps of rule occurrence, we discovered all instances of every rule occurred within 200 milliseconds from initial malware execution with an overall average 12 milliseconds, the strong bond rules averaged 11 milliseconds for $f1$ and 14 milliseconds for $p1$. We randomly selected 120 malware samples and executed them on the analysis platforms Anubis and GFI SandBox. The resulting infection tree structures from both were compared to the MiT's created by MiTCoN. In 114 samples both platforms included nodes (either a process or file) that was excluded in our MiT. Further analysis revealed these nodes represented files and processes belonging to standard Windows process operations and were not part of the malware infection and are therefore false positives. A typical scenario was the inclusion of services.exe, which is used in Windows each time a process requests creation or start of a Windows service. Our approach to MiT building based on fundamental malware infection seems to create more relevant MiTs with the ability to exclude files and processes belonging to standard Windows operations. The high frequency, early occurrence and lack of false positives makes our rules for building MiTs highly effective in analyzing malware.

We test the possible usability of MiTCoN in system disinfection with our previously randomly chosen 120 samples and use the resulting MiT to attempt disinfection of the system. Testing was done in VMWare Workstation running a snapshot of a fresh install of Microsoft Windows Vista logged into the default administrator user account. Kaspersky anti-virus [6] scanned the infected system to assess how successful our disinfection attempt was. Before testing, we invoked a complete system scan by Kaspersky which resulted with no infections found. This was done to assure an initial malware free testing environment and any discovered infections occurring after this initial scan was attributed to the malware executed by us in the test system. Our evaluation was performed in two rounds. Round one of testing was as follows: 1.Initially, the clean state snapshot is loaded 2.A malware sample is copied to the Windows desktop 3.MiTCoN is executed using the sample's path as input 4.The sample is executed for 3 minutes 5.The resulting MiT is saved for later use 6.The infected snapshot is scanned with Kaspersky & the results saved 7.Return to step 1 with next sample. The first round of testing was performed to create MiTs for each of our test samples and to as-

sure Kaspersky can detect the malware infection. In every case, Kaspersky detected the malware. Knowing Kaspersky can detect our malware sample's infection was a pre-requisite to the second round of testing where Kaspersky is used to assess the effectiveness of our disinfection attempt. Round two of testing was as follows: 1.Initially, the clean state snapshot is loaded 2.A malware sample is copied to the Windows desktop 3.The sample is executed for 3 minutes 4. The system is manually disinfected using the samples's MiT from round one 5.The snapshot is scanned with Kaspersky & the results saved 6.Return to step 1 with next malware sample. The main purpose of round two was to assess how effective our disinfection attempt was using a MiT. In each case the files and processes listed in the MiT that were found in the infected system were removed. Kaspersky did not detect any malicious objects in the second round of testing implying our MiTs were effective in eradicating infection from the system. In every case, the details in the MiT sufficed to erradicatd the files and processes from the system.

Limitations. Our testing was conducted in a virtual machine which forcibly excluded using vm-aware malware. MiTCoN is limited by the number of implementations in which a rule can be traced in a specific OS. We are continuously adding new OS specific implementations of a rule into MitCoN as well as discovering new rules reflecting malware infection strategies. Secure testing on actual machines is also being crafted for future use.

5 Conclusion & Future Work

We have presented an abstract approach to building comprehensive MiTs based on rules describing execution events essential to malware infection strategies of files and processes. We developed a MiT creation tool, named MiTCoN, for the Windows Vista platform and tested with 5800 known malware samples. Our analysis revealed MiTCoN was very efficient in MiT building and our rules were highly effective in identifying the relevant malware infection files and processes with high occurrence rate in all samples completing in under 200 milliseconds. In contrast to similar systems, our MiTs avoid false positives by correctly excluding non-malware related processes and files. The MiTs produced by our tool during analysis produced 100% successful manual system disinfection verified with a post-disinfection malware scan. Our results suggests our abstract approach using a malware infection strategy basis for rule creation produces MiTs which are highly effective, improve analysis and disinfection, and produce minimal false positives. Our future work includes adding new execution event rules, secure testing in a real machine and the ability to create MiTs for malware samples that are not WIN32 executables.

Acknowledgements. This work is partially supported by grants from AFOSR, ONR, AFOSR MURI, and the State of Texas Emerging Technology Fund.

References

[1] L. Adleman. An abstract theory of computer viruses. In *CRYPTO '88: Advances in Cryptology*, pages 354–374. Springer, 1988.

[2] F. Cohen. *A Short Course on Computer Viruses*. Wiley Professional Computing, 1994. ISBN 0-471-00769-2.

[3] E. Filiol. *Computer Viruses: from Theory to Applications*. IRIS International series, Springer Verlag, 2005. ISBN 2-287-23939-1.

[4] http://www.gfi.com/malware-analysis-tool/.

[5] G. Hoglund and J. Butler. *Rootkits: subverting the Windows Kernel*. Addison Wesley Professional, 2005.

[6] Kaspersky anti-virus. http://www.kaspersky.com.

[7] N. Kawaguchi, H. Shigeno, and K.-i. Okada. Detection of silent worms using anomaly connection tree. In *Proceedings of the 21st International Conference on Advanced Networking and Applications*, AINA '07, pages 412–419, Washington, DC, USA, 2007. IEEE Computer Society.

[8] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 351–366, Berkeley, CA, USA, 2009. USENIX Association.

[9] J. A. Morales, P. J. Clarke, Y. Deng, and B. G. Kibria. Identification of file infecting viruses through detection of self-reference replication. *Journal in Computer Virology Special EICAR conference invited paper issue*, 2008.

[10] V. Skormin, A. Volynkin, D. Summerville, and J. Moronski. Prevention of information attacks by run-time detection of self-replication in computer codes. *Journal of Computer Security*, 15(2):273–302, 2007.

[11] System service dispatch table. `http://en.wikipedia.org/wiki/System_Service_Dispatch_Table`.

[12] P. Szor. *The Art of Computer Virus Research and Defense*. Symantec Press and Addison-Wesley, 2005. ISBN 9-780321-304544.

[13] R. Vieler. *Professional Rootkits*. Wrox Press, 2007.

[14] Zwxx routines in windows kernel. http://msdn.microsoft.com/en-us/library/ms804352.aspx.