Digital Investigation 22 (2017) S48–S56

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2017 USA — Proceedings of the Seventeenth Annual DFRWS USA

Insights gained from constructing a large scale dynamic analysis platform

Cody Miller [a], Dae Glendowne [b], Henry Cook [c], DeMarcus Thomas [b, *], Chris Lanclos [b], Patrick Pape [b]

[a] Babel Street, 1818 Library St., Reston, VA, USA
[b] Distributed Analytics and Security Institute, 2 Research Boulevard, Starkville, MS, USA
[c] Green Mountain Technology, 5860 Ridgeway Center Parkway, Suite 401, Memphis, TN, USA

Keywords:
Malware
Dynamic analysis
Cuckoo sandbox

A B S T R A C T

As the number of malware samples found increases exponentially each year, there is a need for systems that can dynamically analyze thousands of malware samples per day. These systems should be reliable, scalable, and simple to use by other systems and malware analysts. When handling thousands of malware, reprocessing a small percentage of the malware due to errors can be devastating; a reliable system avoids wasting resources by reducing the number of errors.

In this paper, we describe our scalable dynamic analysis platform, perform experiments on the platform, and provide lessons we have learned through the process. The platform uses Cuckoo sandbox for dynamic analysis and is improved to process malware as quickly as possible without losing valuable information. Experiments were performed to improve the configuration of the system's components and help improve the accuracy of the dynamic analysis. Lessons learned presented in the paper may aid others in the development of similar dynamic analysis systems.

Introduction

As the arms race between malware creators and security professionals progresses, new adaptations are needed, and made, every year. One such adaptation on the security side is malware behavior identification via dynamic analysis. AV-Test indicates that the total number of new malware has increased significantly in the past five years from under 100 million in 2012 to over 500 million in 2016. Due to this increasing number of new malware distributed annually, dynamic analysis systems must be able to process tens of thousands of malware samples per day. In order to meet such a large quota, and remain manageable, the systems need to be reliable, scalable, and convenient for the users. In the effort to create such a system, there are many turning points at which a decision impacts performance and efficiency. It is important to consider all the viable options for these turning points, which can be an overwhelming task for an already complex system. The research presented in this paper seeks to improve understanding of these choices by presenting our system design, and the decisions that were made to ensure high performance and quality of dynamic analysis.

The main software component of the system described in this paper is the open-source dynamic malware analysis platform Cuckoo Sandbox developed by Guarnieri et al. (2013). Due to Cuckoo Sandbox's broad range of virtualization software support and customization (as a result of it being open source and of its extensive plugin support), a multitude of customized systems can be developed around it. The system presented in this paper is just one such iteration, optimized and backed up by testing options at various decision points. Some of the optimization areas focused on in this research include: identifying performance bottlenecks, efficient machine/software configurations for virtualization, and malware execution time limit tradeoffs. Along with the optimization solutions, we also discuss how the presented system is scalable due to our distribution scheme and database storage. Along with detailed explanations of the above areas, within the context of our developed system, this paper provides some lessons learned throughout the process which can aid in the future development and improvement of similar systems.

The growth of malware samples found each year puts more expectations on an already taxing responsibility as a digital

* Corresponding author.
E-mail address: dmt101@dasi.msstate.edu (D. Thomas).

http://dx.doi.org/10.1016/j.diin.2017.06.007

forensics examiner. The only way that digital forensics examiners will be able to respond to the enormous amount of malware being developed is through more sophisticated tools that are developed through lessons of past and current tools. The first contribution of this research is the scalable dynamic analysis platform, which could be used by digital forensics examiners to respond to the sheer amount of malware being developed yearly. Secondly, this paper discusses the different experiments that were used to optimize the platform. Lastly, in the process of developing the scalable dynamic analysis platform, lessons were discovered that should be taken into consideration by future digital forensic tool developers. The lessons could be just as important as the scalable tool because of the ever changing digital environment.

Related work

According to Kruegel (2014), three main aspects of a dynamic analysis system must be true for it to be effective: visibility, resistance to detection, and scalability. Visibility is the ability to effectively monitor the activity of a sample in an analysis environment. With the increased environmental-awareness of malware samples, a sandbox must also be effective at hiding their presence to avoid identification. In addition to these, the ability to process samples at a high rate is a requirement due to the sheer volume of malware samples being produced. Kirat et al. (2011) compare the processing and restore speeds for varying analysis environments, and compare the number of samples that can be processed within a minute. In their experiments, they compared results from BareBox, VirtualBox, and QEMU for automated analysis of 100 samples. The results show the ability to run 2.69, 2.57, and 3.74 samples per minute respectively when the samples were executed for 15 s each. Blue coat malware analysis s400/s500 is a sandbox solution that states the ability to process 12,000 samples daily.

When considering a dynamic analysis system, you must also consider the method that tools use to extract information. Guarnieri et al. (2013) suggested that most systems will monitor the API calls (systems calls) to obtain an idea of what may be occurring in the system. In addition to this, some systems will also monitor the steps between API calls (Kruegel, 2014), perform taint analysis to monitor information as it propagates through the system (Song et al., 2008), execute samples multiple times with varying OS's and system configurations to identify environment sensitivity (Song et al., 2008; Provataki and Katos, 2013), execute hardware emulation (Kruegel, 2014), use bare-metal systems to avoid evasive techniques (Kirat et al., 2011; Kirat et al., 2014), incorporate integration with memory analysis frameworks (Guarnieri et al., 2013), etc. Also, when considering an analysis system, the pros and cons of open-source vs. closed-source projects must be evaluated.

An additional aspect to consider for any dynamic analysis environment is selecting an optimal time of execution per sample. Keragala (2016) stated that samples can exhibit stalling behavior to defeat time-out limits of some systems if not properly selected. Several works did not state a specific execution period, but analysis reports showed execution times ranging between 2 and 3 min (Provataki and Katos, 2013; Vasilescu et al., 2014; Rieck et al., 2011). Lengyel et al. (2014) selected an arbitrary duration period of 60 s. To our knowledge, there has only been a single published work (Kasama, 2014) which performed an empirical evaluation of the optimal execution time for samples in a dynamic analysis system. However, this work had a limited number of samples (5,697) and captured API calls. The experiment in this paper is meant to confirm the results presented by Kasama.

System overview

Cuckoo Sandbox

Cuckoo Sandbox is an automated dynamic analysis sandbox created by Guarnieri et al. (2013). Cuckoo allows the submission of files to be run in an isolated environment. Cuckoo first reverts a VM to a base snapshot (one that is not affected by malware), then it runs the malware on the VM. While the malware sample is running, Cuckoo collects information about what it does in the sandbox such as: API calls, network traffic, files dropped, etc. Cuckoo's collected information will be referred to as a Cuckoo sample for the remainder of this paper. Spengler created Spender-sandbox (Cuckoo 1.3), a modified version of Cuckoo 1.2 that adds a number of features and bug fixes. “Cuckoo modified” (a branch separate from the main Cuckoo version branch) was selected over Cuckoo 1.2 because the modified version adds, among other things, new hooks and signatures.

Cuckoo nodes

To process malware, five hosts running ESXi 5.5.0 were used. The hardware varies slightly between the hosts, two of them have 16 physical cores and three of them have 20 physical cores and all five have 128 Gib of RAM. Each host also has an adapter connected to an isolated network that the hosts share. A virtual machine (VM) running CentOS 7 and Cuckoo (i.e., a Cuckoo node) is on each host and has 64 Gib of RAM and 28 virtual cores. The Cuckoo nodes each have 20 Cuckoo agent VMs within them. All together there are 100 Cuckoo agent VMs managed by the five Cuckoo nodes. This setup of VMs inside of a VM was chosen because, according to Kortchinsky (2009) and Wojtczuk and Rutkowska, though unlikely, malware can escape the agent VM and attack the host; keeping the agent inside another VM adds another layer of isolation.

Cuckoo agents

The driving research behind implementing this system focuses primarily on malware that targets the 32-bit version of Windows 7. Therefore, Windows 7 32-bit virtual machines were used for the Cuckoo agents; QEMU version 2.5.1 was used as the virtualization architecture. The agent virtual machines have: a new installation of Windows 7, 512 Mib of RAM, 1 CPU core, Adobe Reader 11, Python 2.7 installed, and have Windows firewall and UAC disabled. All the agent VMs used the network adapter on the node that is connected to the isolated network of VM hosts.

INetSim

All the agent VMs were connected to an isolated network that does not have access to the Internet. INetSim was created by Hungenberg and Eckert and was used to spoof various Internet services such as DNS, HTTP, and SMTP. It runs on its own VM which is on the same network as the agent VMs. Gilboy (2016) stated that INetSim can improve malware execution as it can trick malware that require the Internet. However, this does not help if the malware needs external resources, such as a command and control server, as INetSim does not provide external (Internet) resources to the isolated network.

Results server

The results server is a VM that provided a way to get the Cuckoo samples from the Cuckoo nodes directly without using Cuckoo's built-in API to fetch the results, thus improving transfer and

processing speed. Each of the Cuckoo nodes has an NFS-connected drive that maps Cuckoo's storage (the directory where Cuckoo places Cuckoo samples) to a mount point on the results server. The results server compresses the samples and transfers them to long term storage. The long term storage is not on the isolated network, therefore making it possible to process the results using resources outside the isolated network.

Database

To improve the stability and efficiency of the system, a database was used to store the malware and to act as a central location in the malware processing pipeline. The database was used to track the malware as it transitions through the following states: submission, Cuckoo processing, and complete. When the malware was submitted to the database, it was marked as being ready for Cuckoo processing. Along with the binary, other task keeping information was submitted, such as hashes, destination on long term storage, current status ('queued' initially), and the priority of the sample. The database also allowed the system to be expandable; other systems can connect to the database to get information about samples and append new information.

Scaling cuckoo

In order to process as much malware as our hardware can handle, our Cuckoo system needed to be scalable. Cuckoo provided a distribution utility to handle scaling. We chose to extend this script to add additional features and flexibility. Our version adds, among other things, the ability to update our database with the status and details of the sample, compress the sample to long term storage, connect to Cuckoo nodes on different subnets, and also fixed some scalability issues. The extended version we created for this purpose ran on the result server and used the existing Cuckoo API and mounted storage paths to compress and store the Cuckoo samples on long term storage. The script retrieved binaries stored on the database and submitted them to the Cuckoo nodes. It does this by keeping track of how many samples a Cuckoo node was processing and how many were already processed, using Cuckoo's REST API. The distribution script supported any number of Cuckoo nodes as long as they are running Cuckoo's API and have mounted the storage path on the results server. While the long term storage's bandwidth is not saturated, additional Cuckoo nodes can be added.

The distribution script submitted 25 samples to each node; when a sample was submitted, it was marked as 'processing' in the database. The script submitted a sample with a random filename that did not contain the letters 'r', 's', or 'm' (so that it cannot contain names of most hashes, such as MD5 or SHA), this was done because Singh expressed that some malware may check its own filename before executing fully. The script monitored the samples using Cuckoo's API, on each Cuckoo node, to determine when a sample had finished processing (meaning Cuckoo has generated reports and is done with the binary). When a Cuckoo node finished processing five samples, five more were added. The number of samples queued on the Cuckoo nodes was small (the number of agent VMs plus five) so that samples with a high priority are not put in a large queue with low priority samples. When a sample was finished, the Cuckoo sample was moved to long-term storage. The binary file, memory file, Cuckoo reports, and dropped files of the Cuckoo sample were compressed as they were moved. The sample is then marked as 'complete' in the database. If the sample failed at any point in the process, the error was stored and the sample was marked as failed in the database. Samples that encountered an error were manually inspected to determine the cause of the error. The full list of errors we have encountered are detailed in Dynamic analysis issues.

Experiments

Distribution time

This experiment aimed to determine how the distribution script we created affects the overall speed of processing samples. The distribution script saved a timestamp when it started processing a sample and another when it was finished. This experiment used the difference of these two times to determine the time added to each sample. This time delta started when the sample was completed in Cuckoo and ended when the sample was on long-term storage.

Experiment

The time delta was calculated for 118,055 samples in the database. This experiment was designed to test the speed of the distribution and not to explore the Cuckoo samples generated. Therefore, the dataset used in this experiment contains all our processed samples from various datasets gathered between 2014 and 2016. All the samples were completed using the distribution script.

Conclusion

Fig. 1 displays a histogram of the time deltas for the samples. Most of the samples take between 50 and 150 s with an average of 114 s. The distribution script can process up to 60 samples in parallel which means, on average, the time per sample is 1.9 s. We consider this to be an acceptable amount of time for each sample.

Cuckoo machinery

The purpose of this experiment was to determine which machinery is more efficient to use on the Cuckoo nodes. The machinery is the architecture that will run the Cuckoo agents. Cuckoo 1.3 (Spengler's version of Cuckoo 1.2), supports physical, KVM, VirtualBox, VMware, ESXi, vSphere, and XenServer. Physical machines were not chosen because we have servers that can support more than one agent at a time; using the server for one agent would waste resources. We chose not to use ESXi, vSphere, and XenServer because they host the Cuckoo agents directly, removing one layer of isolation. Several issues with VMware, such as VM corruption, and speed, formed primary inspiration for this experiment; we wanted to test if QEMU would be a more stable choice. This test was originally performed using VMware 10 and QEMU 0.12. During testing several of the VMware 10 Cuckoo agents became corrupt and had to

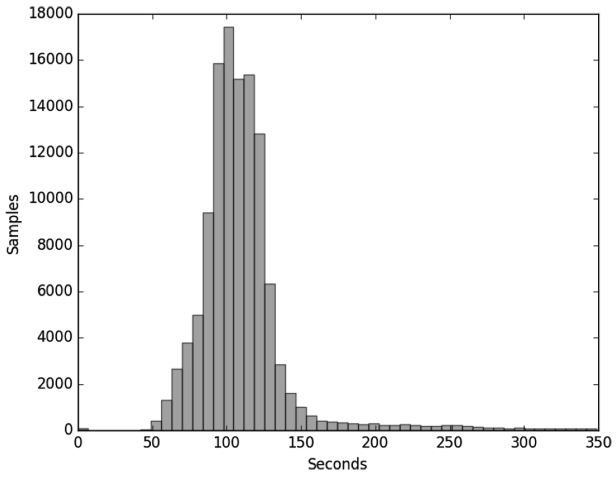

Fig. 1. Distribution time taken for 118,055 samples.

be replaced regularly. This test was performed again with VMware 12 and QEMU 2.5.1.

Experiment

To determine the performance of VMware and QEMU, 20,000 samples from VirusShare.com were processed with Cuckoo on both platforms with 20 agent VMs. VirusShare.com is a service created by Roberts that provided many anti-virus detections for uploaded malware samples. The actual samples were not the focus of this experiment rather the stability of the virtualization architecture running Cuckoo's agents. Each host VM was configured to be identical (same OS, system resources, and configurations). Cuckoo was also configured identically, on both platforms: to only run the samples and not process the results (no reports were generated), and to not dump the agent VM memory. Using VMware, Cuckoo processed the samples in 70 h 6 min (6847 samples per day). Using QEMU 2.5.1, Cuckoo processed the samples in 30 h 14 min (15,862 samples per day). VMware crashed three times during the processing of the 20,000 samples. These crashes caused Cuckoo to stop processing samples and had to manually be restarted. The times between each crash were removed from the total time.

Conclusion

QEMU ran the 20,000 samples 2.3 times faster than VMware and was more stable. Due to these results and QEMU being a free and open source solution, QEMU was chosen as the virtualization technology for our Cuckoo implementation.

Best execution timeout

In Cuckoo sandbox, there was a timeout option that sets the maximum time a sample is allowed to run. After this timeout was hit, Cuckoo terminated the analysis. Cuckoo kept monitoring the malware beyond this timeout while it waited for the processes to terminate if terminate_processes was set in the Cuckoo configuration. If terminate_processes was not set, Cuckoo ended the analysis immediately. The larger this timeout was, the longer the samples could run; however, the higher this value, the fewer processed samples per day. This is due to some malware never terminating itself or looping for an extended period of time. While Guarnieri et al. (2013), Kirat et al. (2011), Lengyel et al. (2014), and others gave a default value for this timeout, they rarely explained why they chose this particular value.

The cumulative percentage of calls was calculated using the enhanced section of the Cuckoo report. Cuckoo generated this section by grouping API calls that are similar. For example, MoveFileWithProgressW, CopyFileA, and DeleteFileW are grouped together in the 'file' group. Table 1 lists each of the eight groups and some example API calls. The full list of groups and their corresponding API calls can be found in the Cuckoo-modified GitHub under modules, processing, and then behavior.py. The enhanced section was chosen over the raw API calls because it requires less transformations to display visually. The cumulative percent was calculated by determining the percent of calls executed at each second, for each enhanced group, by all malware in the dataset.

This experiment used the calls of malware on a single processing of the sample. This does not mean that the malware executed along all code paths. However, a single execution of the malware gives a starting point in determining the timeout.

Experiment

The samples in this experiment were analyzed using Cuckoo with a timeout of 5 min and with Cuckoo set to not terminate processes. Due to the longer timeout, a smaller unique dataset containing 30,346 samples gathered from VirusShare.com was used. This dataset contained a random sampling of malware between 2014 and 2016. Cumulative percentage of API calls per second was calculated on the dataset. After 125 s all the enhanced groups completed at least 90% of their calls. By 1132 s, 100% of all the groups' calls were completed. A graph of the cumulative percent is shown in Fig. 2.

Conclusion

When choosing this timeout you need to decide what loss of information you can afford (if any) to get the performance you desire. Based on this information 125 s was choose for our Cuckoo timeout.

This experiment also had the side effect of providing details about what kind of API calls happen at different times during the execution of malware. According to this experiment, SetWindowsHook and FindWindow APIs are most often called during the first few seconds. Most service APIs (start, control, and delete) are called 20 s after execution. File operations typically get called last, 40 s after the malware started.

Anti-VM

This experiment aided us in our decision to use QEMU over VMware. Various popular anti-VM techniques were used to determine how well the VM detection worked on Cuckoo agent VMs. For this experiment, non-sandbox related detections (for example Cuckoo detections) have been ignored as they are not in the scope of the experiment. While hardening the agent VMs was not the focus of this experiment; it is something to consider when deciding to use QEMU or VMware.

Experiment

Pafish (Deepen), an open source project focused on identifying sandboxes and analysis environments, used common malware techniques to fingerprint systems. It was run via Cuckoo on both QEMU 2.5.1 and Vmware Workstation 12 and detected the following virtualization identifiers on both QEMU and Vmware:

Table 1
Enhanced groups.

Group	Description	Example APIs
file	File based operations such as copy, read, delete, write, move	CopyFileA, DeleteFileA, NtWriteFile
dir	Directory based operations such as delete, create	RemoveDirectoryA, CreateDirectoryW
library	Loading and using libraries (such as DLLs)	LoadLibraryA, LdrLoadDll, LdrGetDllHandle
windowname	Retrieving handle to a window	FindWindowA. FindWindowExA
registry	Registry based operations such as set, create, read, delete	RegSetValueExA, RegCreateKeyExA, RegDeleteKeyA
windowshook	Windows API hooking	SetWindowsHookExA
service	Service based operations such as start, modify, delete	StartServiceA, ControlService, DeleteService

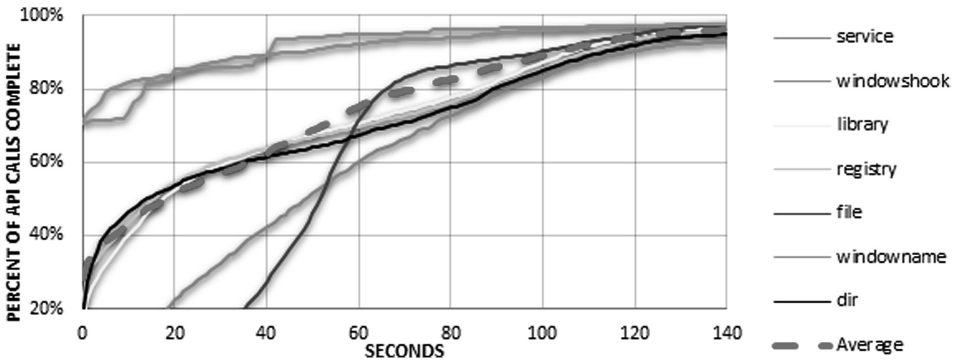

Fig. 2. Cumulative percent of enhanced groups. The dashed line represents the average cumulative percent for all enhanced groups.

- The correct CPU vendor ("AuthenticAMD" for QEMU and "GenuineIntel" for Vmware) and Windows version (6.1 build 7601)
- A "VM CPU" by checking the RDTSC CPU timestamp counter and by checking the hypervisor bit in the CPUID.
- Under 60 Gib disk, under 2 Gib RAM, and less than 2 CPU cores

Pafish also detected the following for Vmware:

- A Vmware device mapped SCSI registry (HKLM/HARDWARE/DEVICEMAP/Scsi/Scsi Port 1/Scsi Bus 0/Target Id 0/Logical Unit Id 0 "Identifier")
- A Vmware MAC address starting with 00:50:56
- A Vmware WMI Win32 bios serial number

Conclusion

Some of the VM detections had simple fixes, such as the VMware registry (by altering the key), MAC (by changing the MAC), and disk/RAM/CPU cores (by increasing these resources). Others required more research to determine how these detections can be concealed. The results of this comparison showed that QEMU had less detectable virtualization through basic detection techniques. This conclusion provided another reason to choose QEMU over VMware for our system.

Hardware specification

This experiment aimed to give an estimate of the hardware resources required to run 20 Cuckoo agent VMs using QEMU. When the Cuckoo nodes were first setup, the number of resources to give them was unknown. The resources were overestimated and each Cuckoo node was given 64 Gib of ram and 28 CPU cores, which was an over-estimation by a significant amount.

Experiment

A Cuckoo node was monitored while it processed samples on all 20 Cuckoo agent VMs. It was determined that on average the agents used a fourth of the RAM they were given and cuckoo's processing utility used 2 Gib RAM per parallel process. QEMU used CPU cores no greater than half the number of agents running. Similarly, the processing utility used total CPU cores no more than half the number of parallel processing configured.

Conclusion

The equation defined here use the following key terms:

- R: minimum Gib RAM required to give each Cuckoo node without over-commiting RAM
- C: Gib of RAM to dedicate to Cuckoo's main process
- O_R: Gib of RAM to give the OS and other programs
- O_C: CPU cores to give the OS and other programs
- P: number of parallel processes Cuckoo's processing utility uses
- $agent_{count}$: number of agents on each node
- $agent_{ram}$: Gib of RAM to give each agent

$$R = C + O_C + P \times 2 + \frac{agent_{count} \times agent_{ram}}{4} \quad (1)$$

Equation (1) was derived to estimate the minimum Gib of RAM to give each Cuckoo node. It allows Cuckoo's main process to have RAM dedicated to it, 2 Gib dedicated to each process of Cuckoo's processing utility, and a fourth of the maximum RAM the agents can use. The main Cuckoo process is not memory intensive and it is recommended to provide it no more than 1 Gib. For our system, we choose $O = 4$ Gib.

$$R = C + O_R + \left\lceil \frac{agent_{count} + P}{2} \right\rceil \quad (2)$$

To estimate the minimum number of CPU cores each Cuckoo node needed, Equation (2) was derived. This allowed Cuckoo to have dedicated cores and the agents and process utility to have cores equal to half of the number of threads they create.

Improving execution

This experiment was designed to determine if, by changing specific things in the agent VM, the malware would run differently or have more compatibility with the agent. This experiment targets malware that needs a framework or checks for previous user activity/system usage as described in Deepen and Willems.

Experiment

Ten Cuckoo agents were used to process a dataset containing 10,000 randomly chosen samples gathered from VirusShare.com between 2015 and 2016. The dataset was run on the agents with different agent configurations using Cuckoo 1.3. The base configuration was the same agent listed in Section Cuckoo Agents. The hardened configuration was the base configuration with the following additional changes:

Added Documents

- "My Documents" has 5 JPGs, 1 txt, 5 PDFs, and 3 data files
- "My Music" has 3 MP3s
- "My Pictures" has 6 JPGs and 1 GIF
- "My Videos" has 4 MP4s

New Programs

- Firefox 38.0.5, Notepad++ v7, VLC 2.2.4, 7-Zip 16.02, Adobe Flash player 10.1.4, Java 6.45

New Frameworks

- Microsoft Visual C++ 2005, 2008, 2010, 2012, 2013, and 2015 redistributable
- Microsoft .NET 3.5 and 4.6.1 frameworks

Recent Documents/Programs

- All the added documents were opened multiple times. Each new program was run multiple times.
- Running Programs
- Windows explorer
- Notepad
- All update services for new software were disabled

Many of these changes were added as a result of the previous experiments and lessons learned from them. These changes were made to make the agent appear as if it has been/is being used by an actual user. The agent was left running for 30 min while programs and documents were manually opened and closed. The running programs were still running when Cuckoo started analysis on the malware. To evaluate the differences of each configuration, specific items in the Cuckoo sample of each configuration were compared. API calls of the malware were inspected to determine if the malware executed differently and the network activity was examined to see if the malware operated differently over the network.

Conclusion

Of the 10,000 samples, 9014 ran completely on the base configuration compared to 9421 that ran on the hardened configuration. A complete run was achieved when the malware executed properly and Cuckoo was able to capture at least one API call. Samples that did not completely run on both the normal and hardened were removed, leaving 8834 samples. The 1166 samples that did not have a complete run were examined and it was determined that 363 samples immediately exited with no hooked APIs called (the malware ran properly but decided to exit), 474 had a Cuckoo error unrelated to the sample, and the reason for the remaining 329 could not be determined. Of the 10,000 samples, 23 of them required an external DLL that was not on the virtual machine. By having frameworks installed 705 more samples were able to run completely. 63% of these were Microsoft .NET and 37% were Microsoft Visual C++.

Table 2 shows the changes in the types of API calls performed by the sample in the base configuration versus the hardened one. Call type is the label Cuckoo groups API calls by. Percentage of samples is the percent of samples that had an increase in API usage. Average percent increase is the percentage of the average amount that the call type APIs increased or decreased usage by. Maximum percent increase is the maximum amount that the call type increased by.

On the hardened configuration the samples had the following differences:

- 54.98% of the samples exhibited an increased number of unique API calls. The average increase of these samples was 7.88.
- 60.22% of the samples had more total API calls, 10.61% had fewer, and 29.17% had the same amount.
- 89.28% of the samples ran for a longer duration.
- There were no new IP addresses or domains requested. However, some samples made different network calls, though there was no substantial difference as only 2.91% of the malware did so.

Table 3 displays the differences between the two configurations in terms of number of files and registry reads, writes, and deletes, number of mutexes created or accessed, and number of Cuckoo signatures flagged. All actions except reading keys had no significant impact between the two configurations as their average usage increased by a small amount. This small increase could be due to a number of things unrelated to the configurations.

The results of this experiment clearly show that samples on the hardened configuration ran differently than on the normal configuration. As expected, more malware was able to run due to required frameworks being installed on the hardened configuration. This experiment also shows that malware on the hardened configuration executed more code. Further research is required to determine the exact reason the malware behaved differently.

Table 2
Changes in types of API calls in the base versus the hardened configurations.

Call type	Percentage of samples	Average percent increase	Maximum percent increase
reg	32.82%	24.63%	85.35%
misc	37.96%	8.95%	90.77%
process	24.11%	5.37%	86.44%
file	30.58%	5.23%	88.03%
sleep	41.1%	2.23%	92.73%
network	7.79%	1.70%	23.43%
socket	6.55%	1.67%	42.63%
reg_native	28.40%	1.34%	86.77%
crypto	4.10%	1.09%	91.18%
special	30.39%	0.70%	48.05%
window	26.26%	0.43%	90.45%
sync	35.64%	0.42%	36.06%
thread	37.74%	0.37%	17.02%
services	10.81%	0.30%	26.86%
other	0.00%	0.00%	0.0%

Lessons learned

Virtualization architecture

Lesson: *Choose an appropriate dynamic analysis platform.* There are many dynamic analysis platforms available. Egele et al. (2012) provide an extensive survey of dynamic analysis systems and techniques. This serves as a good starting point for identifying many of the current systems and the techniques they use. We considered several dynamic analysis systems that have appeared in the literature.

We initially filtered the list of dynamic analysis systems based on their availability and level of active development. First, a system that was strictly web-based, requiring us to submit samples to a remote server to receive results was not considered acceptable for the purposes of this work as the goal of this work was to construct a scalable platform for analyzing large quantities of malware. Therefore, a version of the system must be available for download either in source code or binary form so that it could be installed and operated locally.

Table 3
Differences in configuration (base minus hardened).

Name	Minimum increase	Maximum increase	Average increase	Samples increased	Samples decreased
Read Key	−2744	3231	309.09	3982	985
Read File	−41	250	4.62	3947	716
Write Key	−30	2516	1.24	1615	265
Write File	−27	159	1.04	2103	136
Mutexes	−83	124	0.73	2537	554
Delete Key	−7	52	0.49	1116	41
Delete File	−7	328	0.34	1618	52
Signatures	−6	14	0.19	1549	295

Second, the system must be under active development. Malware analysis is a constantly evolving area requiring consistent effort to stay relevant. Some of the systems we considered had source code available, but had not been updated in five or more years.

Third, the system should be freely available.

- **Anubis** (Analysis of unknown binaries) – web-based; currently offline
- **CWSandbox** (Willems et al., 2007) – commercial
- **Norman Sandbox** (Norman sandbox analyzer) – commercial
- **Joebox** (Joebox) – web-based
- **Ether** (Dinaburg et al., 2008) – not updated since 2009
- **WILDCat** (Vasudevan and Yerraballi, 2006, 2005, 2004) – no source code/binary
- **Panorama** (Yin et al., 2007) – no source code/binary
- **TEMU** (Song et al., 2008) – not updated 2009

PANDA and Cuckoo Sandbox both met these criteria. PANDA (Dolan-Gavitt et al., 2015), captures all instructions within the agent VM at the virtualization architecture level (outside the agent operating system). This low-level instruction data provides everything Cuckoo Sandbox provides; however, it is in a raw format. PANDA was not as mature as Cuckoo at the time and lacked plugins to convert this raw data into the information made readily available by Cuckoo. It also lacked the robust reporting engine within Cuckoo. PANDA plugins could have been created to accomplish this, but due to time constraints, Cuckoo Sandbox was chosen.

After deciding on Cuckoo, we had to make a decision about which version of Cuckoo Sandbox we would use. The three primary choices available at that time were Cuckoo 1.2 (Guarnieri), Cuckoo 2.0rc1 (Guarnieri), and Cuckoo-modified (Spengler) (a.k.a. Cuckoo version 1.3), which is a modified community version of Cuckoo Sandbox 1.2. We did not choose 2.0rc1 because at the time it had just been released and was still in the process of fixing a number of serious bugs and we required something more stable. However, Cuckoo 2.0rc1 does offer some benefits over the other two choices as far as functionality goes and, for future work, we may consider using a Cuckoo 2.0 build. The key factor when choosing between 1.2 and 1.3 was that 1.3 provided support for some features that we required from the sandbox. These features, as stated by Spengler, are detailed below:

- Fully-normalized file and registry names
- Correlating API calls to malware call chains
- Anti-anti-sandbox and anti-anti-VM techniques built-in
- The ability to restore removed hooks
- Additional signatures and hooks
- Automatic malware clustering using Malheur
- MIST format converter for Cuckoo logs

All of these features were provided in version 1.3, whereas only a subset of the functionality was available in version 1.2. Version 1.2 provided limited options for dealing with malware that use anti-sandbox techniques; whereas version 1.3 provided a significant amount of anti-anti-sandbox techniques. Version 1.3 provided many more signatures than 1.2 and the signatures available in version 1.3 were more sophisticated than those present in version 1.2. Over the duration of our project, the number of available signatures in version 1.3 has outgrown what is available from version 1.2, and also includes a number of signatures that were included in the 2.0rc1 release.

Dynamic analysis issues

Lesson: *Check and truly understand your analysis*. Cuckoo has some stability issues that cause Cuckoo samples to be inconsistent between runs. We encountered issues where running a sample once through Cuckoo would produce an invalid Cuckoo sample. However, running the same sample again would produce a valid Cuckoo sample. An invalid Cuckoo Sample could have any of the following issues: missing files, missing data in the report, corrupt reports, and/or Cuckoo errors. These issues led us to thoroughly check Cuckoo Samples at different stages. In this process, the Cuckoo samples were checked, and invalid samples were marked as invalid in the database. The invalid samples can then be deliberately queued again in another run.

Our first iteration resulted in losing over 10% of Cuckoo samples due to various errors. Some of these issues were due to bugs in the distribution script, which were then corrected. Some of the errors were solved by switching from Cuckoo version 1.2 to 1.3; however, not all of the errors have been resolved. In a dataset of 24,463 samples gathered from VirusShare.com, there were 899 errors of this type (3.67% of samples). After manual inspection of the errors, the following explanations for the errors were found:

Error: Report.json missing processes or process calls

- *Missing files* – (164 samples) these samples needed external resources such as a DLL.
- *Missing frameworks* – (14 samples) these samples needed a framework, such as MSVCP70, MSVCP90, MSVCP100, MSVCP110, or Cygwin that was not installed.
- *Different OS* – (131 samples) these samples called Windows API functions that were different from the agent's Windows 7 OS. These samples caused a Windows error: invalid procedure entry point.
- *Corrupt/Invalid* – (2 samples) these samples caused Windows errors not a valid Windows 32 bit application or cannot be run in 32 bit mode. Note that all these samples had a valid PE header and a PE_TYPE header option value of 267 (32-bit).
- *Application crashed* – (18 samples) these samples crashed on startup causing a Windows application crash.

Error: Cuckoo error – The package "modules.packages.exe" start function raised an error: Unable to execute the initial process, analysis aborted.

- *Corrupt/Invalid* – (217 samples) these samples caused Windows errors not a valid Windows 32 bit application or cannot be run in 32 bit mode. Note that all these samples had a valid PE header and a PE_TYPE header option value of 267 (32-bit).
- *Side-by-side configuration* – (10 samples) these samples caused Windows error: side-by-side configuration incorrect

Error: Cuckoo error system restarted unexpectedly

- *Restarts* – (7 samples) in Cuckoo 1.3, if the malware restarts the VM Cuckoo will only capture a partial execution.
- *Unknown* – (35 samples) these samples did not restart the VM upon manual inspection, it is unknown why Cuckoo raised this error.

Error: Cuckoo error – The analysis hit the critical timeout, terminating.

- *Unknown* – (256 samples) These samples appeared to be error free, but it is possible that they are not complete analysis.

Error: Various other errors

- *False errors* – (80 samples) Some errors, such as missing Cuckoo sample files, may have been caused by things other than Cuckoo.

When these samples were run through Cuckoo again, they did not produce errors.

Another issue we experienced was that some of our tests used what Cuckoo called duration in the reports. We determined later that this number does not actually represent the duration that the malware ran. Instead, this duration represented the time Cuckoo took to completely process the sample, which includes time before/after the malware started/ended. The actual malware duration we used was the time between the first API call and the last API call. We also found that in some API call sequences there would be a large gap in time. This was due to Windows updating the system time after Cuckoo started the analysis. This was a confirmed bug within Cuckoo. The solution is to disable NTP on the guest.

Improving analysis performance

Lesson: *Disable unnecessary functionality*. There are several processing modules that Cuckoo has enabled by default that we found to be unnecessary. The following modules were disabled because they slow the Cuckoo nodes and their features can be calculated using the Cuckoo Sample at a later date: memory, dropped files, static, and strings modules. Another tweak, one recommended by the Cuckoo developers, was to use Cuckoo's processing utility. By default the main Cuckoo process (cuckoo.py) handles submitting the samples to the agents and processing the results. We found that cuckoo.py became unstable and its performance dropped while samples were submitted and when the results were processed. This issue was fixed by separating the processing using Cuckoo's processing utility script: processing.py.

Database

Lesson: *Use a database*. Using a database provided a simple way to automate sample processing. Originally, samples were manually submitted to Cuckoo from a directory which made it hard to keep track of processed samples. Now samples are added to the database and automatically submitted to Cuckoo. In the development of this tool the use of Couchdb, a NoSQL database, was used for a couple of reasons. The first being the large volume of data that we would have and the second being the ability to scale out the architecture which is an important aspect of this platform. However, the database did not solve all our data issues; since most of the Cuckoo sample files are large, it is troublesome to store them in the database. This led us to create links in the database to these files on the shared file system. This meant that every machine/analyst that processed these files also had to be connected to the file system. Another issue is that each piece of our system had to know the exact structure of the database and changing the structure meant changing each system.

Future work

Now that thousands of samples are automatically processed per day, there are a number of additional improvements that can be made. First the submission of samples for Cuckoo generation is currently a manual process. In the future, a REST API will be constructed that handles much of the sample submission. Cuckoo generation would not need to know the exact structure of the database as long as it follows the API guidelines.

Currently, the system only supports Windows 7 32-bit. In the future, any Cuckoo supported OS and additional agent configurations will be added. This could be done by setting a flag in the database for which agent the sample should be run on and then the distribution script will submit to the correct Cuckoo node and agent. Specific agents will be configured to have a different set of programs installed or be completely different OS/architectures.

Moving forward, there is also a need for a processing component for extracting information from each Cuckoo sample. A scalable, extendable plugin framework will be developed to allow feature extraction from any part of the Cuckoo sample. This framework will use the same database as the Cuckoo generation process, and will be able to automatically extract features from completed samples. Once the processing framework is complete, we will use the features extracted from Cuckoo samples to support future machine learning classification and clustering work.

Conclusion

In conclusion, we developed a dynamic analysis platform, performed various experiments on the platform to optimize it, and provided valuable lessons we have learned throughout the process. The experiments that were performed on this platform not only helped make key decisions in the platform's development, but also provided insight into best practices/choices for virtualization software and machinery behind other dynamic analysis systems. The dynamic analysis platform presented is able to process any number of malware samples per day, only being bound by the computational resources available. The platform submitted and processed samples to and from a database, allowing other systems to be connected. Other systems can share information about samples and append to a sample record. This allows further, more complex, sample analysis.

References

Analysis of unknown binaries [online].

AV-Test. Malware statistics [online].

Blue coat malware analysis s400/s500 [online].

Deepen, D. Malicious Documents Leveraging New Anti-vm and Anti-sandbox Techniques [online].

Dinaburg, A., Royal, P., Sharif, M., Lee, W., 2008. Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, ACM, pp. 51–62.

Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R., 2015. Repeatable reverse engineering with PANDA. In: Repeatable Reverse Engineering with PANDA. ACM Press, pp. 1–11. http://dx.doi.org/10.1145/2843859.2843867.

Egele, M., Scholte, T., Kirda, E., Kruegel, C., 2012. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. (CSUR) 44 (2), 6.

Gilboy, M.R., 2016. Fighting Evasive Malware with DVasion (PhD thesis). University of Maryland.

Guarnieri, C., Tanasi, A., Bremer, J., Mark, S. The cuckoo sandbox.2013 URL https://www.cuckoosandbox.org/.

Guarnieri, C. Cuckoo Sandbox 1.2 [online].

Guarnieri, C. Cuckoo Sandbox 2.0–RC1 [online].

Hungenberg, T., Eckert, M. INetSim. URL http://www.inetsim.org/.

Joebox: A Secure Sandbox Application for Windows to Analyse the Behaviour of Malware [online].

Kasama, T., 2014. A Study on Malware Analysis Leveraging Sandbox Evasive Behaviors (PhD thesis). Yokohama National University.

Keragala, D., 2016. Detecting Malware and Sandbox Evasion Techniques.

Kirat, D., Vigna, G., Kruegel, C., 2011. BareBox: efficient malware analysis on bare-metal. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACM, pp. 403–412.

Kirat, D., Vigna, G., Kruegel, C., 2014. BareCloud: bare-metal analysis-based evasive malware detection. In: 23rd USENIX Security Symposium. USENIX Association. OCLC: 254320948.

Kortchinsky, K., 2009. CLOUDBURST – a VMware Guest to Host Escape Story. BlackHat, USA.

Kruegel, C., 2014. Full system emulation: achieving successful automated dynamic analysis of evasive malware. In: Proc. BlackHat USA Security Conference, pp. 1–7.

Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A., 2014. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In: Proceedings of the 30th Annual Computer Security Applications Conference, ACM, pp. 386–395.

Norman sandbox analyzer [online].

Provataki, A., Katos, V., 2013. Differential malware forensics. Digit. Investig. 10 (4), 311–322. http://dx.doi.org/10.1016/j.diin.2013.08.006.

Rieck, K., Trinius, P., Willems, C., Holz, T., 2011. Automatic analysis of malware

behavior using machine learning. J. Comput. Secur. 19 (4), 639–668.
Roberts, J.-M. VirusShare [online].
Singh, A. Defeating Darkhotel Just-in-time Decryption [online].
Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P., 2008. BitBlaze: a new approach to computer security via binary analysis. In: International Conference on Information Systems Security. Springer, pp. 1–25.
Spengler, B. Spender-sandbox. URL https://github.com/spender-sandbox/.
Vasilescu, M., Gheorghe, L., Tapus, N., 2014. Practical malware analysis based on sandboxing. In: 2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference. IEEE, pp. 1–6.
Vasudevan, A., Yerraballi, R., 2004. Sakthi: a retargetable dynamic framework for binary instrumentation. In: Hawaii International Conference in Computer Sciences.
Vasudevan, A., Yerraballi, R., 2005. Stealth breakpoints. In: Computer Security Applications Conference, 21st Annual. IEEE, p. 10.
Vasudevan, A., Yerraballi, R., 2006. Cobra: fine-grained malware analysis using stealth localized-executions. In: Security and Privacy, 2006 IEEE Symposium on. IEEE, p. 15.
Willems, C. Sandbox Evasion Techniques – Part 2 [online].
Willems, C., Holz, T., Freiling, F., 2007. Toward automated dynamic malware analysis using cwsandbox. IEEE Secur. Priv. 5 (2).
Wojtczuk, R., Rutkowska, J., Following the white rabbit: software attacks against intel VT-d technology, invisible things lab (ITL), Tech. Rep.
Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E., 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, ACM, pp. 116–127.