Enhancing Automated Malware Analysis Machines with Memory Analysis

Tomer Teller, Adi Hayon

Security Innovation Group
Check Point Software Technologies
{tomert,adiha}@checkpoint.com

***Abstract*—In this paper, we present novel methods based on memory analysis to enhance automated malware analysis machines and boost malware detection rates of executable files.**

***Keywords*—Malware Analysis, Malware Detection, Memory Analysis,**

I. Introduction

In the last year, we have witnessed a plethora of malicious samples that would render signature and heuristics based-detection completely useless. This includes non-persistent, volatile payloads that operate only in memory, obfuscated / packed samples that decrypt themselves on-demand during run-time and evasive user-mode / kernel-mode rootkits that subvert the automated malware analysis machines. The increasing amount and diversity of malicious samples render manual malware analysis inefficient, time consuming and not scalable. Proliferation of this threat forced security professionals to automate the malware analysis process with Automated Malware Analysis Systems. These systems analyze a potentially malicious sample using a mixture of techniques to detect malicious activity, mainly Static Analysis and Dynamic Analysis.

Static Analysis:

The act of analyzing a given sample without executing it to get a better understanding of what it does. This includes but is not limited to checking the sample's MD5, Import Hash[2] and fuzzy hash values against VirusTotal as well as checking if the sample is similar to another sample that the analyst has already seen in the past by disassembling its functions and using binary diffing techniques.

Dynamic Analysis:

The act of analyzing a given sample by executing it inside a controlled environment for a configurable amount of time, observing its behavior and flagging suspicious/malicious activities. When time is up, the sample residues are collected from the controlled environment, processed and analyzed. This includes actions such as API call sequence analysis, control-flow/data-flow analysis and dynamic binary instrumentation (DBI).

Current Analysis Limitations:

While these analysis methods are proven, malware authors introduced new evasion techniques to render them unsuccessful. Obfuscators, Cryptors and Packers are just some of the tools attackers have in their arsenal to evade static analysis. These tools mutate, obfuscate and pack code sections to look different from any other sample, modify the entry point to make it hard to reverse engineer, insert anti-debugging tricks to make it challenging to debug and destroy import tables to blur the sample intentions. These techniques[4] pushed malware researchers towards dynamic analysis, that is, instead of fighting with the sample statically, they simply run it in a controlled environment and observe its execution. Unfortunately, malware authors prevailed and introduced new tricks in that field as well. Sleeping ("running out the clock"), detecting the underlying virtual/analysis machine and user-interaction which forces a human to click on dialogs or move the mouse are just some of the evasion techniques malware employ today to dodge dynamic analysis. In general, during dynamic analysis the concept of "what you see is what you get" applies. If the malware "feels comfortable" on the machine, it will execute its malicious logic, if not, it may mislead the researcher and act benign. With dynamic analysis, code coverage may depend on input given to the sample throughout its execution. While analyzed in an automated system, this input is minimal.
While these evasion techniques are becoming a bigger challenge for security researchers, this paper does not cover solutions to these issues.
Most dynamic-based analysis systems are based on a predefined list of API call sequences. Each sequence defines a certain behavior in the system. To illustrate how API call sequence detection works, consider the following example:

Example 1. An automated malware analysis machine executes a sample and monitors all the API calls that occurred in a system. It then detects the following 3-gram sequence {WriteProcessMemory,CreateRemoteThread,LoadLibrary} (with specific parameters). This sequence indicates a DLL injection into a process and is a known trick used by malware to remain persistent on the machine. At this point, the system may raise a flag since a known malicious behavior was observed during execution.

Unfortunately, there exists a plethora of techniques that malware authors introduced in order to evade API call sequences including but not limited to:

1) User-mode / Kernel-mode rootkits can hook and subvert WinAPI calls, thus bypassing the entire API call sequence mechanism [5]
2) Calling undocumented functions that achieve the same malicious behavior and are not listed in the predefined list of API call sequences
3) Calling non-hooked native functions
4) Custom WinAPI function implementations

These limitations led malware analysts to include Memory Analysis into the investigation process.

II. Memory Analysis

Memory Analysis is the act of analyzing a snapshot of the physical memory image at some point(s) in time to determine the overall state of a computer. Since every OS object ends up on the RAM at some point, this is a great place to look for malicious artifacts.
In-memory data includes but is not limited to:

- Processes and Threads (current and terminated)
- File Handles
- Network Objects (open TCP/UDP connections)
- Modules / Drivers
- Caches
- Windows Registry keys

By analyzing memory objects, Memory Analysis can help:

- Discover system inconsistencies that might indicate a rootkit
- Collect hidden artifacts that cannot be retrieved using the OS-provided API (e.g. user passwords / cookies)
- Identify system activity and overall machine state
- Pinpoint malware that operates in-memory only

Digital Forensics and Incident Response (DFIR) practitioners use commercial and open-source tools to extract these digital artifacts from the memory image (RAM) and detect malicious activity. These tools use techniques such as pool tag scanning and file carving to retrieve data and do not rely on API calls that can be subverted. While it may seem like memory analysis is a great tool for dissecting malware, it comes with a price and some disadvantages.

Memory Analysis Limitations

- The process of running memory analysis tools is manual and does not scale
- Interpreting output from tools requires deep knowledge in OS internals
- Anti-Forensics tools exist [8] to prevent memory acquisition [7] and analysis as well as plant fake artifacts in-memory to decoy the investigation [6]
- Artifacts from a single memory dump lack context, since there is no baseline memory dump to compare it with. It is difficult to make meaningful conclusions without information about when the artifact was created, modified, deleted, etc
- Taking memory dumps requires accurate timing. If we take it at the wrong time, we may "miss the action", that is, malicious artifacts may not exist yet or already disappear from memory

Current Memory Analysis Automated Approach

Today, automated solutions that perform memory analysis execute a suspicious sample in a controlled environment and take a memory dump for offline investigation when the sample terminates. Since memory is volatile, this approach risks "missing in on the action" as malicious artifacts may appear and disappear intermittently. A possible solution to this problem can come in a form of interval-based memory dumps.

Possible Solution: Interval-Based Memory Dumps

In this approach, we configure the controlled environment to grab a memory dump in intervals. While this process may yield better results, it has the huge disadvantage of being random, that is, the time the memory dumps are taken is arbitrary and the memory dump itself may not contain the malicious artifacts that the analyst was looking for as they may slip in-between the memory dump intervals.
To illustrate this problem, consider an automated system that runs a malicious sample and is configured to take a memory dump every 30 seconds:

[time 00:00:00] Malware is executed
[time 00:00:05] Allocates memory in a remote process
[time 00:00:07] Writes code to the allocated region
[time 00:00:15] Executes the code
[time 00:00:22] Free the code
[time 00:00:30] Memory Snapshot is taken

Since the malware cleaned the evidence from memory before the snapshot was taken, no malicious artifacts could be found. This example is one of many that allow an advanced malware to evade automated memory analysis using current tools and techniques. It is clear that during dynamic analysis, when taking memory dumps, timing is critical to discover all the suspicious artifacts.

III. Trigger-Based Memory Dumps and Differential Analysis

We would like to propose a new technique to solve the timing issues and enhance automated analysis systems. We call this technique "trigger-based memory dumps". It works by defining "interesting" actions or triggers, that happen in the system during the sample execution and signal the analysis machine that "it's a good time to take a memory dump".

Triggers

At the time of writing, we implemented the following triggers:

1) API-Based: A set of malicious behaviors that can be identified via API call sequences (e.g. Process Injection, Autostart capabilities, Hook installation, etc.).
During execution, we monitor all the API-calls in the system and in case we detect one of the behaviors, we trigger a memory dump. It is possible to limit the number of memory dumps taken for every detected API sequence.

2) Performance-Based: A utility that monitors the processes and CPU utilization by sampling the performance counters and detecting abnormal deviations in their usage. In case a predefined, configurable threshold was reached, we trigger a memory dump.[1]

3) Instrumentation-Based: We trace the sample execution and monitor it for abnormal mathematical computations (e.g. unpacking/decryption/decoding loops) using dynamic binary instrumentation techniques. In case a predefined, configurable threshold was reached, we trigger a memory dump.

System life cycle

Preprocessing : This process occurs only once during the lifetime of the system and is part of the implementation. In this phase we revert the controlled environment to a clean state and grab a baseline memory dump to be used during differential analysis (described later).

Sample Processing: For each new sample that arrives, the system will:

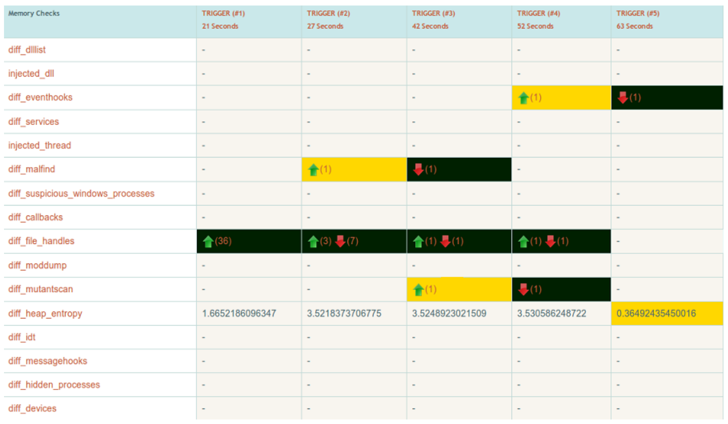
1) Revert to the clean snapshot of the controlled environment
2) Execute the sample in the controlled environment
3) Trace and monitor the sample execution while detecting the aforementioned triggers
4) When a trigger is detected:
    a) Suspend the controlled environment
    b) Grab a memory image dump
    c) Resume the controlled environment
5) Before the sample terminates or execution time exceeded:
    a) Suspend the controlled environment
    b) Grab a final memory image dump
    c) Terminate execution
6) Stop the controlled environment

At the end of this process, the system ends up with multiple memory dumps.

Differential Analysis

The next phase is to analyze the differences between the memory dumps that were taken during the execution. These differences include artifacts that were added, removed and/or modified in-memory during the time the snapshot was taken. In order to achieve this, for each memory dump we run different memory analysis plug-ins that extract malicious artifacts from it and save the results for later processing.
Next, the results are sorted in a descending manner based on their respective memory dump generation time.
The system will compare each consecutive result set pair, look for artifacts that were added, removed and/or modified and write the differences to a file.

[1]It is important to note that using this approach may cause false positives if the threshold is incorrectly configured. In the context of this trigger, we define a false positive as a memory dump that we triggered and does not contain any new malicious artifacts.



Memory Checks	TRIGGER (#1) 21 Seconds	TRIGGER (#2) 27 Seconds	TRIGGER (#3) 42 Seconds	TRIGGER (#4) 52 Seconds	TRIGGER (#5) 63 Seconds
diff_dlllist	-	-	-	-	-
injected_dll	-	-	-	-	-
diff_eventhooks	-	-	-	⬆(1)	⬇(1)
diff_services	-	-	-	-	-
injected_thread	-	-	-	-	-
diff_malfind	-	⬆(1)	⬇(1)	-	-
diff_suspicious_windows_processes	-	-	-	-	-
diff_callbacks	-	-	-	-	-
diff_file_handles	⬆(36)	⬆(3) ⬇(7)	⬆(1) ⬇(1)	⬆(1) ⬇(1)	-
diff_moddump	-	-	-	-	-
diff_mutantscan	-	-	⬆(1)	⬇(1)	-
diff_heap_entropy	1.6652186096347	3.5218373706775	3.5248923021509	3.530586248722	0.36492435450016
diff_idt	-	-	-	-	-
diff_messagehooks	-	-	-	-	-
diff_hidden_processes	-	-	-	-	-
diff_devices	-	-	-	-	-

Figure III.1. An example of a (partial) time-line report generated after differential analysis

At the end of this process, a time-line report (Figure:III.1) is generated which highlights all the malicious artifacts and their modifications that were discovered in-memory. The combination of trigger-based dump and differential analysis allows us to detect subtle changes that happen in-memory during execution and allows us to detect advanced evasive malware that may appear in-memory at some point during the execution and then disappear, thus, evading traditional memory analysis.

IV. Implementation Details

Based on the analysis flow outlined above, this section provides a detailed discussion of our techniques implementation inside cuckoo sandbox[1], an open source automated malware analysis machine. While the details in this section are specific to cuckoo sandbox, all of them may be trivially applied to other analysis machines.

Modified Cuckoo Sandbox

When Cuckoo receives a sample for analysis, it executes it inside a controlled environment such as VirtualBox, QEMU or VMWare. We chose to use QEMU due to its emulation capabilities and Virtual Machine Introspection support. We modified Cuckoo to grab a clean memory image dump of the virtual machine and save it on disk before execution so we can later compare it with other memory dumps. This process occurs only once and is crucial as the controlled environment may contain multiple unrelated artifacts and it is important to differentiate between the residues of a new executing sample and the artifacts that were there before. We extended the hooking capabilities of CuckooMon to facilitate our logic. Mainly, we added code that suspends an offending process once an "interesting" action has occurred (as defined in Section 3 "Trigger-Based Memory Dumps"), grabs a memory dump of the entire RAM / Process involved while saving it on disk and resuming execution. In case the malware tries to terminate prematurely, we added a kernel hook on ZwTerminateProcess to block its termination, take an extra, final memory dump and terminate. We then added code that analyzes each one of the dumps by running multiple Volatility plug-ins (Malfind, apihooks, psxview, etc.) including our own custom plug-ins (described later) and compute the differences starting from the final memory dump down to the clean snapshot. Each of the differences is written into a Cuckoo's reporting module's JSON file.

Custom Plug-ins

For the purpose of our research we wrote a couple of memory analysis plug-ins that are suitable for differential analysis:

Antivirus strings Plug-in: Malicious samples may try to detect if a certain Antivirus (AV) is present on the controlled environment prior to executing its malicious logic. Once detected, the sample may act differently on that machine (Self-Terminate, act benign, try to disable the AV, etc.). The proposed plug-in dumps all readable strings from memory and checks for popular AV string names. This plug-in runs on each memory dump as the AV strings may be encrypted at some point during the execution and decrypted during a certain trigger.

Process heap entropy Plug-in: Advanced Malware may try to encrypt/compress files in-memory during packing/exfiltration process. The proposed plug-in computes the entropy of each process heap and checks if it was changed above a certain configurable threshold value and if so, the size of the change. This plug-in runs on each memory dump as the entropy may increase at a certain point and then decrease.

Modified PE Header Plug-in: Once loaded, advanced Malware may modify its own PE header [3] in a way that detection tools will not be able to dump or detect them. The proposed plug-in monitors injected DLLs and verifies that their PE headers were not modified during execution.

V. Case Study

Consider the following Zbot dropper sample (MD5: 75ab3481fe83335b6c58867c12be0c51) caught in-the-wild.
This sample is packed and obfuscated which makes it harder to analyze statically.
During run-time, the sample unpacks itself in chunks using the following sequence:

- Allocates memory with PAGE_READWRITE permissions using VirtualAllocEx
- Writes its payload to the allocated memory region using WriteProcessMemory
- Adjusts the allocated memory region permissions to PAGE_EXECUTE_READWRITE using VirtualProtectEx
- Executes the payload

Once the payload was executed and in order to avoid memory forensic analysis, the malware overwrites the allocated memory region with zeros and reduce its permissions back to PAGE_READWRITE.
The malware keeps reusing that region while covering its tracks until it is no longer needed, at that point it calls VirtualFreeEx to free the allocated memory region.
Let's review the memory analysis of this sample using the following 3 methods.

Classic Memory Analysis

In this method, we configured the controlled environment to run the sample for 3 minutes and grab a memory dump at the end of execution.
When the malware terminated, the system ended up with one memory dump.
We then ran multiple tools and plugins to analyze the memory dump and could not find any trace of the allocated memory region.
This makes sense since the malware cleaned up its tracks and removed the allocated memory region before we took the memory dump. It is clear that taking extra memory dumps during the malware execution will yield better results.

Interval-Based Memory Analysis

In this method, we configured the controlled environment to run the sample for 3 minutes and grab a memory dump every 30 seconds.
When the malware terminated, the system ended up with 6 memory dumps.
We then ran multiple tools and plugins to analyze the memory dumps and we did see the allocated memory region, however it contained zeros and it was not executable which means it won't raise any suspicions.
Depending on the interval configuration it is possible that we would have seen the malware payload in the memory region before it was wiped, however, there is no single interval that would fit all samples. It is clear that a mechanism that grabs memory dumps during "interesting" actions is required.

Trigger-Based Memory Analysis

In this method, we configured the controlled environment to run a sample for 3 minutes and grab a memory dump using our proposed trigger-based approach (we limited the number of memory dumps per trigger to one).
The controlled environment was configured with API-Based / Performance-Based / Instrumentation-Based triggers and the malware was executed. While the Performance-based/Instrumentation-based didn't yield interesting results, the API-based trigger approach showed insightful results. During execution, the malware hit two of the predefined API sequences {VirtualAllocEx , WriteProcessMemory , VirtualProtectEx} and {WriteProcessMemory , VirtualProtectEx} which in return generated two memory dumps. When the malware terminated, the system ended up with 4 memory dumps:

- (D1) A clean memory dump taken before the malware execution (as described in section 3)
- (D2) Memory dump triggered by the first API sequence
- (D3) Memory dump triggered by the second API sequence

- (D4) A final memory dump taken just before the malware terminated (as described in section 3)

We then used differential analysis plugins to analyze the memory dumps.

Comparing D1 with D2: New artifacts were added between D1 and D2. We can see that a new memory region was allocated, written into and its permissions were changed.

Comparing D2 with D3: Artifacts that were added in the previous comparison were modified. Specifically, in D3 the memory region was zeroed out and its permissions were changed.

Comparing D3 with D4: Artifacts that were added in D2 did not appear in D4. This makes sense as the malware called VirtualFreeEx to free the memory region.

Looking at the “story” that the comparisons are telling us, we can deduce the binary behavior and help focus the security researcher on the interesting parts of the malware logic.

VI. Future Work

As part of our future work, we plan to automate the verdict decision (malicious or benign) based on malicious artifacts found during trigger-based analysis.
We currently work on implementing new triggers and converting our trigger-based solution to a plug-in framework so it would be easier to write, test and share triggers with the community.
Also, we plan on adding VMI (Virtual Machine Introspection) support for non-intrusive memory acquisition and optimize the plugin execution times.
We encourage the readers of this paper to integrate our solution in their automated systems, implement new triggers and share them with the community.

VII. Conclusion

Memory Analysis is becoming a great technique to analyze Malware. Unfortunately, the current approach has many limitations. In this paper, we proposed possible solutions to overcome these limitations and enhance the current solutions. Our main contribution is the Trigger-Based memory analysis approach for automatically taking memory dumps when interesting actions happen in the system during execution. This approach gives the security researcher invaluable information of what happens during the execution of the sample and not only what happened at the end. In combination with other techniques for memory analysis, this approach strengthens security and gives more valuable insight in the efforts to detect malicious activity.

Acknowledgments The authors would like to thank the reviewers for their insightful comments.

References

[1] Cuckoo sandbox (http://cuckoosandbox.org/).

[2] Imphash calculation (https://www.mandiant.com/blog/tracking-malware-import-hashing).

[3] "snake" / the uroburus malware (gdata software). 2014.

[4] Barbosa Branco and Neto. Overview of malware anti-debugging, anti-disassembly and anti-vm techniques (http://research.dissect.pe/docs/blackhat2012-paper.pdf). 2012.

[5] J. Butler and K. Kendall. Blackout: What really happened (black hat 2007).

[6] M. Cohen. Anti-forensics and memory analysis (http://scudette.blogspot.co.il/2014/02/anti-forensics-and-memory-analysis.html). 2014.

[7] Milkovic. Defeating windows memory forensics (http://www.slideshare.net/lmilkovic/defeating-windows-memory-forensics). 2012.

[8] J. Williams and A. Torres. Add: Complicating memory forensics through memory disarray (shmoocon, 2014). 2014.