White Paper

McAfee
An Intel Company

Build a Better Sandbox

A working strategy for comprehensive malware protection

Table of Contents

The Coevolution of Malware and Detection Analytics

Cybercriminals and IT security strategists are locked in an escalating arms race. As the malware used to infiltrate IT environments becomes ever more sophisticated and evasive, new technologies are emerging to find the needle in the haystack, no matter how well camouflaged.

One of the most promising developments on the defensive side is occurring in the area of dynamic analysis detection, commonly known as sandboxing. A number of products are currently available and others are in various stages of commercialization. Not surprisingly, the architectural approaches taken by leading developers vary widely, as do the ways these new products fit into larger security strategies. At this time, it's difficult to make confident comparisons between technologies, or even to be sure that basic terminology is consistently applied.

This paper proposes a logical design strategy for dynamic malware analysis that optimizes detection effectiveness, efficiency, and economics. We will attempt to identify the limits of dynamic detection and the supplemental methods necessary to ensure robust security. Finally, we will suggest some important distinctions that are commonly obscured by imprecise terminology.

Dynamic versus static analysis

Perhaps the most important bit of context for any discussion of advanced malware sandbox solutions is the distinction between dynamic and static analysis. Dynamic analysis seeks to identify malicious executable files by loading them into a secure runtime environment, usually virtualized, and observing their behavior for some predetermined interval.

To properly contrast this tactic with static analysis requires that we first resolve a common discrepancy in the latter term's use. True static analysis (sometimes called static code analysis) predicts an executable's probable behaviors based on a detailed assessment of its code. The term 'static analysis' is often misapplied to simpler and less revealing techniques (sometimes called static file analysis) that may only assay a portion of a file's header, or that can access only unobfuscated file content. These have limited utility in identifying advanced malware, and all uses of static analysis in this paper refer to techniques capable of extracting, parsing, and analyzing a file's full code.

Both dynamic and true static techniques have strengths and weaknesses. Dynamic analysis can identify malware with a very high degree of confidence based on direct observation of its behavior. It is the most reliable way to accurately identify hidden threats in complex executables, but can be easily defeated by various stratagems. A file may simply outwait the observation period, delaying the start of any revealing behavior for a predetermined interval that may be longer than an economically viable sandbox inspection. A file may also be programmed to recognize a secure environment by the absence (or presence) of certain resources, and to execute only a limited set of deceptively innocuous operations.

Static inspection identifies malicious code with a lower degree of confidence than dynamic analysis because it relies on inference rather than observation, yet it also provides a window into the nature of latent (non-executing) code to which dynamic analysis is entirely blind. For example, static code analysis identifies structural similarities between latent code and known malware samples. It quantifies the percentage of code that executes during a sandbox evaluation, and even maps the logical execution paths of a complex file without actually running any of the code.

What is striking is the degree to which the strengths and weaknesses of static and dynamic analysis are complementary. While the techniques commonly associated with a malware sandbox are dynamic, it is unlikely that reliable and accurate detection can be achieved unless both dynamic and true static code analyses are applied in a well-integrated process. An effective sandbox must be simultaneously dynamic *and* static.

The performance imperative

One characteristic that dynamic and static code analysis share is that both are computationally intensive, to the extent that neither can be applied to network traffic flows in real time, and both must be applied selectively to avoid degrading network and application performance. It may take several minutes or even hours to complete this type of analysis, depending on the product and vendor. A rational approach is to front-end them with more resource-efficient technologies that can quickly and economically eliminate easily identified threats.

We believe that a highly efficient and effective malware sandbox can be created by layering several types of analytical engines in a stacked sequence of increasing computational intensity. All unknown files intercepted by network security sensors can be referred to this service for evaluation. Each file passes through the stacked inspection engines beginning with the fastest and least resource-intensive. Files that are identified as malicious at each level are immediately blocked and removed from the inspection flow, reducing the load on downstream, more computationally intensive analytics.

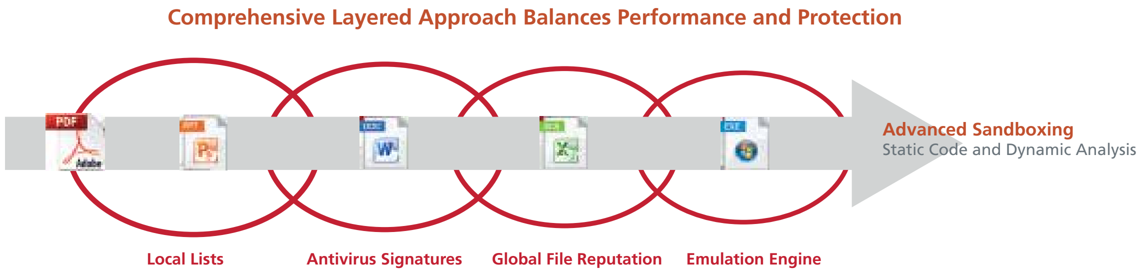

Comprehensive Layered Approach Balances Performance and Protection
Advanced Sandboxing
Static Code and Dynamic Analysis
Local Lists
Antivirus Signatures
Global File Reputation
Emulation Engine

Figure 1. A downselect multiengine malware inspection process.

While the inspection stack composition should be extensible to incorporate new detection techniques as they emerge, today's state-of-the-art can be well represented by a sequence that begins with known attack detection (signatures and reputation services), followed by real-time behavioral detection (heuristics and emulation), dynamic analysis, and static code analysis. We call this a downselect inspection architecture. Let's examine this stratified process sequence level by level.

Level One: Known Attack Detection

The two malware detection techniques used in first-level inspection are the oldest and most widely deployed. They are also some of the most real-time and computationally light techniques. Signature-based inspection, the core technology in all antivirus products, provides fast, positive identification based on pattern matching against a library of known malicious code samples. Reputation services collect intelligence on known sources of previous attacks, including hashes of the actual malware, geographic locations, domains, URLs, and IP addresses, providing a basis for identifying known, unknown, and zero-day attacks arriving from known malicious vectors.

Both of these techniques are fast, computationally frugal, and provide high-confidence threat identification in real time. Their essential attributes are (1) a comprehensive library of known threat signatures and sources and (2), a fast, reliable infrastructure for acquiring new threat intelligence globally and distributing it to local sensors.

Since both of these technologies—and, to a lesser extent, the technologies included in level two (below)—are widely deployed in existing security products, it is extremely helpful when our sandbox management capabilities include the ability to separately define the sandbox inspection engines to be applied to the files referred by each type of sensor. As a result, signature-based inspection that has been performed by an IPS service, for instance, will not be repeated at the sandbox.

Level Two: Real-time Behavioral Defenses

Two separate types of detection are also applied in the second inspection tier—heuristic and emulation. Heuristic identification uses rules and behavior pattern analysis to create generic constructs and distinguish similarities between a suspect file and groups or families of related known threats. Emulation simulates file execution on a stripped-down host environment and logs the resulting behaviors. The emulation environment may include a subset of the CPU, memory, and operating-system API resources. Emulation is sometimes described as halfway-to-full dynamic analysis or sandboxing light, but is sufficiently less resource intensive, allowing it to provide real-time results.

Heuristics and emulation provide real-time identification of previously unobserved threats and are only slightly less reliable than signature-based techniques. They involve some decompilation and unpacking of code, but because this is a real-time process, few facilities are deployed here for unpacking or reverse engineering obfuscated files.

We should note that different, language-specific emulators are needed for different types of content (executables, shell code, JavaScript, HTML, and Java). An emulation engine's reliability and effectiveness are directly related to the completeness of its capabilities.

Level Three: Dynamic Analysis

The third level in our model sandbox architecture marks the dividing line between analyses that takes place effectively in real time and those more resource-intensive techniques that inevitably impose slightly greater latency. This is where files that have not been conclusively identified as malicious in prior inspections are allowed to execute in a safely isolated virtual environment. True dynamic analysis differs from emulation in that it instantiates a fully operational runtime environment that is virtualized and isolated to allow safe execution of potentially malicious code, then logs or classifies all observed behaviors.

Target-specific sandboxing

There are two common approaches to configuring the virtual environments used in a malware sandbox. The differences are important because most IT environments comprise a variety of hardware and software platforms, and most malware samples target a specific operating environment or application. The first approach is to virtualize a single generic environment and use it in all sample analyses. This approach risks missing malicious behaviors that are dependent on specific resource sets or configuration parameters that may be unavailable in the generic image, but it is resource efficient as only one analytical pass is needed.

The second approach is to virtualize multiple environments (various Windows server platforms and configurations, perhaps, plus a selection of PC and mobile platform images). Suspect samples are run against each of these environments. This strategy, however, still risks missing the actual target environment, may produce more false positives, and is many times more computationally expensive as well.

A far more effective and efficient strategy is to run a suspect executable in a virtual environment that exactly matches the system for which the file was targeted. This approach requires that a broad range of operating system options must be available, or that gold images of all endpoint platforms in the environment can be imported. The sandbox must be able to identify the target host environment on the fly and quickly launch a matching VM—and this is not a network activity, but requires integration with endpoint systems. If these conditions can be met, the probability is greatly increased that a suspect file's full range of potential behaviors can be elicited and observed, and an accurate assessment made of its intent.

The importance of interaction

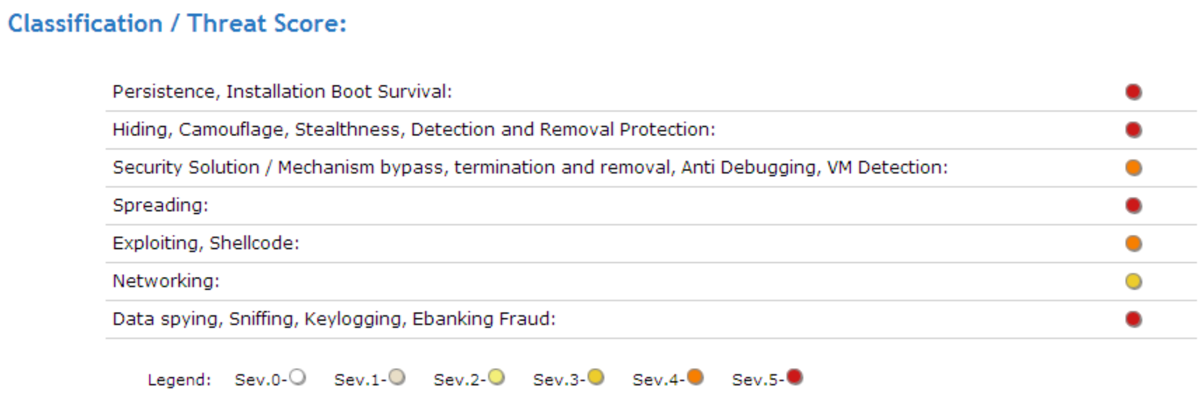
In order for sandbox inspection to fully evaluate an executable's intent, the virtual environment must respond interactively to its behavior just as a normal host system would. In particular, the sandbox must automatically emulate the normal host response to network connection requests. Lack of these expected responses could inform malware that it is being analyzed in a sandbox, allowing it to take evading actions. Reputation services must also be available to the sandbox, so that high-risk requests for access to known malicious IP addresses, URLs, and files can be immediately identified as high-probability threat indicators.

In addition to the interactivity that must be available to automatically triggered inline file inspections, a fully interactive mode should be available to security analysts for offline manual analysis. In this mode, an analyst should be able to manually launch a VM and load an executable sample with full KVM functionality. Often it is only with the ability to launch browser sessions and other normal workplace applications that specific threat behaviors can be triggered and observed.

Interpreting dynamic analyses

The first two levels of our inspection stack produce simple, real-time outputs. An unknown file is quickly identified as a known threat through a signature match, or is seen to be sufficiently malicious in real-time emulation environments. In either case, a binary decision to block or pass is quickly made.

In contrast, the initial output of a dynamic analysis is a log file that only becomes meaningful with correlation. Specific behavioral events must be identified and aggregated, and their significance assessed in the context of other events. The output from an enterprise-ready sandbox is not a long and complicated log readout, nor is it an overly simple pass/block decision. A useful sandboxing tool for the enterprise must provide an aggregated and organized report that calls out and classifies relevant behaviors and awards an overall score. This score may be sufficient to trigger a blocking decision, or may require supporting insight from subsequent static analysis. Either way, it gives actionable information that security operators can use.



Classification / Threat Score:

Persistence, Installation Boot Survival:	🔴
Hiding, Camouflage, Stealthness, Detection and Removal Protection:	🔴
Security Solution / Mechanism bypass, termination and removal, Anti Debugging, VM Detection:	🟠
Spreading:	🔴
Exploiting, Shellcode:	🟠
Networking:	🟡
Data spying, Sniffing, Keylogging, Ebanking Fraud:	🔴

Legend: Sev.0-○ Sev.1-⚪ Sev.2-🟡 Sev.3-🟡 Sev.4-🟠 Sev.5-🔴

Figure 2. A behavioral summary and classification report from dynamic analysis.

Level Four: Static Code Analysis

The final downselect in our expanded multistage sandbox is true static-code analysis, which we might more descriptively call disassembly list-code analysis. This process launches with the stage-three dynamic analysis and it incorporates some outputs of dynamic inspection as they become available.

We mentioned earlier that our proposed stage-two inspection techniques—heuristics and emulation—rely on access to a file's source code, but that few facilities would be provided in that real-time analysis for extracting obfuscated or packed code. That type of deep forensic examination is precisely our objective here in stage four.

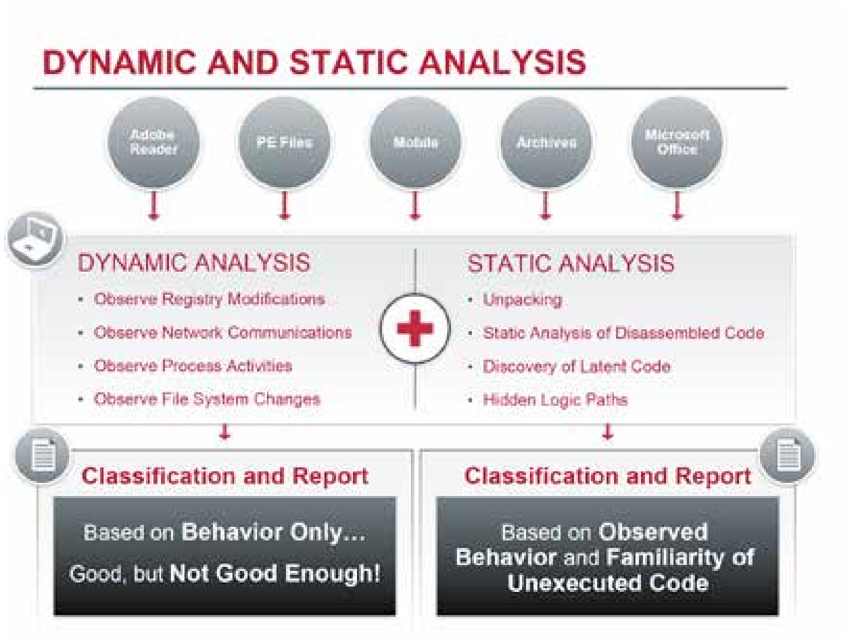
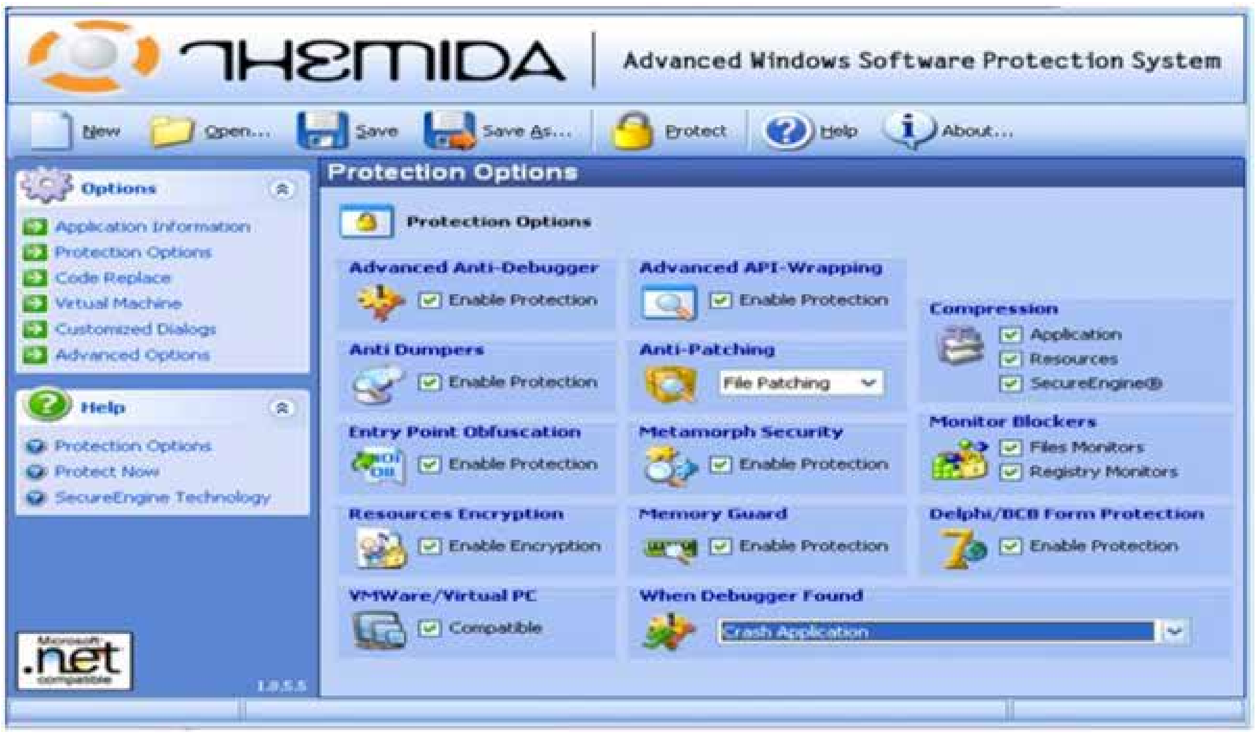

DYNAMIC AND STATIC ANALYSIS
Adobe Reader
PE Files
Mobile
Archives
Microsoft Office
DYNAMIC ANALYSIS
• Observe Registry Modifications
• Observe Network Communications
• Observe Process Activities
• Observe File System Changes
STATIC ANALYSIS
• Unpacking
• Static Analysis of Disassembled Code
• Discovery of Latent Code
• Hidden Logic Paths
Classification and Report
Based on Behavior Only…
Good, but Not Good Enough!
Classification and Report
Based on Observed Behavior and Familiarity of Unexecuted Code

Figure 3. Dynamic analysis alone is an incomplete analysis.

Packing, unpacking, and reverse engineering

There are perfectly legitimate reasons for concealing or obfuscating a program's compiled executable code, the most obvious being intellectual property protection. Software developers understandably wish to prevent competitors from reverse engineering their products by working backwards from distributed assembly code to the source. Not surprisingly, other enterprising developers have created a variety of commercial tools to make that exceedingly difficult. Called packers (for example, Themida, Armadillo), these tools make it easy to apply an arsenal of masking and randomization techniques to compiled program code, making it extremely hard to reconstruct the assembly code and thus access the source. Malware writers or malware developers have simply adopted the software industry's own techniques, making their cloaked attacks much more difficult to separate from legitimate files.

Difficult, perhaps, but not impossible.



Protection Options

Advanced Anti-Debugger: ✓ Enable Protection
Advanced API-Wrapping: ✓ Enable Protection
Compression: ✓ Application, ✓ Resources, ✓ SecureEngine®
Anti Dumpers: ✓ Enable Protection
Anti-Patching: File Patching
Entry Point Obfuscation: ✓ Enable Protection
Metamorph Security: ✓ Enable Protection
Monitor Blockers: ✓ Files Monitors, ✓ Registry Monitors
Resources Encryption: ✓ Enable Encryption
Memory Guard: ✓ Enable Protection
Delphi/BCB Form Protection: ✓ Enable Protection
VMWare/Virtual PC: ✓ Compatible
When Debugger Found: Crash Application

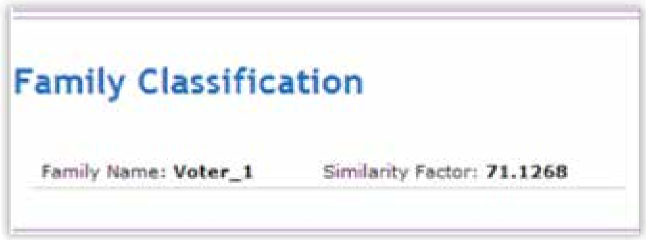
Figure 4. The obfuscation menu in Themida, a powerful packing tool for Windows applications.

In the fourth stage of our malware sandbox, packed and obfuscated files are reverse engineered to recover intact versions of the compiled assembly code. These are then parsed and subjected to statistical analysis, providing:

- An assessment of similarity with known malware families.
- A measurement of the latent code which did not execute during dynamic analysis.
- A logical map of the file's complete execution path(s).

Modern advanced persistent threats (APTs) are typically adaptations of known—and effective—malware code. Minor modifications are sufficient to evade signature inspection, which requires an exact match to convict. However, inspection of the entire code, compared to a library of known malware family references, often uncovers extremely stealthy malware.

For example, a suspicious file that has some minor indicators of compromise which are not severe enough to block the file, but happen to have better than 70% similarity to a known malware family (for example, conficker and voter_1), is a file that should be blocked. Without static code analysis, this malware would have penetrated the network.


Family Classification

Family Name: Voter_1 Similarity Factor: 71.1268

Figure 5. Family similarity is powerful evidence of malicious potential.

In a similar manner, APTs increasingly seek environmental awareness or require a specific sequence of interaction to elicit action. This means that much of stealthy malware's code may remain latent during dynamic analysis. Even if the behavior of this latent code cannot be extracted, the fact that a large portion of a suspect file's code remains latent should be an important part of a security analyst's investigation.

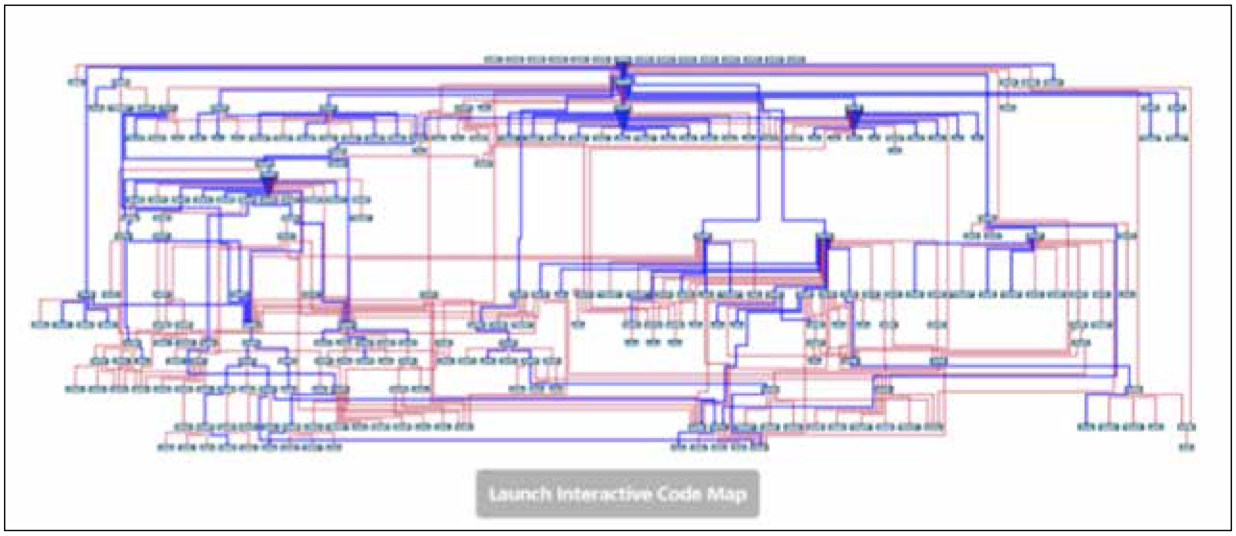
Consider a suspect file that shows no malicious behavior. Dynamic analysis alone would conclude that the file was safe. However, what if the file had greater than 70% similarity to a known malware family, and what if more than 40% of the file's code was not active or analyzed in the sandbox? Those two pieces of circumstantial evidence are enough to block delivery of this file, at least until a security operator can investigate it.

Behavior Summary (57 percent code coverage):

🟠	Hides file by changing its attributes.	🟡	Manipulated with active content in the admin temporary directory.
🟡	Detected executable content dropped by the sample.	🟡	Obtained and used icon of legit system application.
🟡	Created executable content under Administrator temporary directory.	🟡	Created executable content under Windows directory.
🟡	From Microsoft: CreateURLMoniker can produce results that are not equivalent to the input, its use can result in security problems.	🟡	Executed active content from Windows system folder.
🟡	Committed a region of memory within the virtual address space of a foreign process.	🟡	Set callback function to control system and computer's hardware events.
○	Tried to connect to a specific service provider.	○	Downloaded data from a webserver.
○	Created content under Windows System directory.	○	Registered (unregistered) the service name in a Dynamic Data Exchange (DDE) server supports.

Figure 6. Significant portions of latent code point to holes in dynamic-only analysis.

When human investigation is necessary, a diagram of file operations can be an extremely helpful forensics tool. Unlike log files or observations of behavior (dynamic analysis), a diagram helps security operators investigate hidden areas and interaction of the code. This is often the key to illicit otherwise latent code, especially when used in conjunction with a sandbox that allows manual interaction with the suspect file within the VM.



🟦 Blue lines show code that was dynamically executed.

🟥 Red lines show code that was statically executed.

Figure 7. Visualizing file processes allows an operator to stimulate latent code.

These findings are then incorporated with the observations from stage-three dynamic analysis to provide an overall score indicating the degree of certainty that the sample file or executable is malicious.

A Downselect Strategy for Sandbox Superiority

An advanced malware detection service designed and configured following the stacked, downselect strategy described above will provide a significantly more sophisticated, reliable, and cost-effective solution than anything currently available in the marketplace. It will prevent sandbox overloading by screening out known and easily identifiable threats using high-percentage, low-overhead signature-based inspection and reputation intelligence services. It will dramatically increase the efficiency and accuracy of dynamic analysis with target-specific execution sandboxing. Finally, true static code analysis will render transparent the cloak of obfuscation to reveal the hidden nature of latent and evasive code.

For once in the convoluted history of malware-security coevolution, security is about to make a proactive leap forward.

Learn more at www.mcafee.com/atd.

McAfee
An Intel Company

2821 Mission College Boulevard
Santa Clara, CA 95054
888 847 8766
www.mcafee.com