Malware Dynamic Analysis Evasion Techniques: A Survey

Shehryar Kamran

Abstract

In dynamic analysis we have two modes, manual mode which will use debuggers and automatic mode which uses sandboxes for inspection of malware. Creators of malware evade dynamic analysis (manual as well as automatic) techniques by targeting present analysis techniques, revising their techniques by making it more advance. In this survey-paper we will compare different ways of evasion with respect to dynamic analysis and at the end purpose that how they get evaded against different analysis techniques.

INTRODUCTION

Malware dynamic analysis is the technique which provide us the information what it does by inspecting malware on run time. Due to lack of static analysis functionality, dynamic analysis came into being but malware creators use some advance techniques which will prevent this analysis, either its manual mode or automatic mode both get effected from malware evasion technique (Chiu, 2015). We will compare different solutions which will help in dynamic analysis of evasion techniques for malware both modes with their pros and cons, last but not least our recommendation.

Literature Review

The increase of 33% from past in malware variants are defeating AV because the signature of samples is not present (Awards, 2022). Dynamic analysis of malware is very crucial process because if malware detects the environment that system is having tools or is an isolated environment then it will destruct itself, detection of system is done by using fingerprinting technique (Oyama, 2018). On the other hand, if we run on real system, it will destruct the system like removing system files from C drive (Anish, 2012). To analyse malware, command and control server communication we will decompile it in assembly language, monitor network traffic, convert byte-code to assembly code, later on into high-level code (Oleg Kulchytskyy, Anton Kukoba, 2021). This technique is also known as reverse engineering where we know that malware have anti-debugger or obfuscation security (Reverse engineering VertexNet malware, 2015).

To analyse that is also important because sometime our machine plays a role of bot which take commands from bot master and perform some malicious activity like DDos attack (Barlow, 2000).

A transparent system, who exposes a smaller number of system properties will be able to tackle the problem of evasion. A good sandbox can have the properties of scalability, visibility, resistance to detection (Kruegel, 2014). Advance level sophistication can only be done when we know how inside of malware are working (Hornat, 2007).

Honey monkey systems are used by big companies like google to test malicious pages, the attack on client side make this technique honey client evasion which lead to more vulnerable attacks (Nadji, 2010).

Some malwares do not need special equipment to test them, example of such kind of malware is Win32/Industroyer which will interrupt the working of industrial control system (Cherepanov, 2012).

One method to make a user run a file is to make it look genuine and when user run them, they will execute malicious script in memory directly which make them in fileless malware category. This type of PowerShell based script malware were 13 percent from total gathered malware in 2017 (Rozena, 2018).

Methodology

Sandboxes:

Classification and Comparison of Malware Sandbox Evasion Techniques

<table>
<tr><th colspan="2">Criteria</th><th rowspan="3">Complexity</th><th rowspan="3">Pervasiveness</th><th rowspan="3" colspan="2">Efficacy Level</th><th colspan="2">Sandbox Countermeasure Tactics</th><th rowspan="3">Detection Complexity</th><th rowspan="3">Example</th></tr>
<tr><th colspan="2"></th><th rowspan="2">Complexity</th><th rowspan="2">Effectiveness</th></tr>
<tr><th>Cat.</th><th>Tactic</th></tr>
<tr><td rowspan="5">Detection-Independent</td><td rowspan="2">Stalling</td><td rowspan="2">Low-Medium</td><td rowspan="2">Medium</td><td rowspan="2" colspan="2">All Architecture</td><td colspan="2">Sleep Patching</td><td rowspan="2">Very High</td><td rowspan="2">[104, 105, 106, 107]</td></tr>
<tr><td>Low</td><td>Low</td></tr>
<tr><td rowspan="2">Trigger-Based</td><td rowspan="2">Low</td><td rowspan="2">Medium</td><td rowspan="2" colspan="2">Emulation-Based, Bare-metal</td><td colspan="2">Path Exploration</td><td rowspan="2">Moderate</td><td rowspan="2">[49, 111, 112, 113, 18,]</td></tr>
<tr><td>High</td><td>Moderate</td></tr>
<tr><td>Fileless (AVT)</td><td>High</td><td>Low</td><td colspan="2">All Architectures</td><td colspan="2">N/A</td><td>Very High</td><td>[61]</td></tr>
<tr><td rowspan="6">Detection-Dependent</td><td rowspan="2">Fingerprinting</td><td rowspan="2">High</td><td rowspan="2">High</td><td rowspan="2" colspan="2">VM-Based, Hypervisor-based, Emulation-Based</td><td colspan="2">Using heterogeneous analysis, Artifact Randomization</td><td rowspan="2">Moderate</td><td rowspan="2">[83, 84, 87]</td></tr>
<tr><td>Moderate</td><td>Moderate</td></tr>
<tr><td rowspan="2">Reverse Turing Test</td><td rowspan="2">Medium</td><td rowspan="2">Medium</td><td rowspan="2" colspan="2">VM-Based, Hypervisor-based, Bare-metal,</td><td colspan="2">Digital Simulation, path exploration</td><td rowspan="2">Moderate</td><td rowspan="2">[97, 98]</td></tr>
<tr><td>Low</td><td>Low</td></tr>
<tr><td rowspan="2">Targeted</td><td rowspan="2">Very High</td><td rowspan="2">Low</td><td rowspan="2">VM-Based, Hypervisor-based,</td><td rowspan="2">Emulation-based</td><td colspan="2">Path exploration</td><td rowspan="2">Very High</td><td rowspan="2">[96, 102, 104]</td></tr>
<tr><td>High</td><td>Low</td></tr>
</table>

Anti-Debugging:

Classification and Comparison of Malware Anti-Debugging Techniques

Cat.	Tactic	Technique (Criteria)		Complexity	Resistance	Countermeasure Tactic	Pervasiveness	Malware Sample	Efficacy -Level
Detection-Dependent	Fingerprinting	Reading PEB	IsDebuggerPresent() CheckRemoteDebuggerPresent()	Low	Low	Set the Beingdebugged flag to zero	Very high	[54, 132]	1
				Medium	Low	Set heap_groawable glag for flags field and forceflags to 0			1
			NtGlobalFlags()	Low	Medium	Attach debugger after process creation			1
		Detecting Breakpoints	Self-scan to spot INT 3 instruction Self-integrity-check	Low	Medium	Set breakpoint in the first byte of thread	High	[85, 87]	1, 2
			Read DR Registers (GetThreadContext() etc.)	Low	Medium	Reset the context_debug_registers flag in the contextflags before/after Original ntgetcontextthread function call			1, 2
		System Artifacts	FindWindow(), FindProcess(), FindFirstFile(),	Low-High	Low-High	Randomizing variables, achieve more transparency	Medium	[59]	1, 2, 3
		Mining NTQuery Object	ProcessDebugObjectHandle() ProcessDebugFlags() ProcessBasicInformation()	Medium	High	Modify process states after calling/skipping these API	Medium	[58, 116, 137]	1, 2
		Parent Check	GetCurrentProcessId() + CreateToolhelp32Snapshot()+ (Process32First())+Process32 Next()	Medium	Medium	API hook	Low	[59]	1, 2
		Timing-Based Detection	Local Resource: RDTSC timeGetTime(), GetTickCount(), QueryPerformanceCounter GetLocalTime() GetSystemTime()	Low	High	Kernel patch to prevent access to rdtsc outside privilege mode, Maintain high-fidelity time source, Skip time-checking APIs	Medium	[62, 87, 89]	1, 2, 3, 4
			Query external time source (e.g. NTP)	Medium	N/A	None, open problem			
	Traps	Instruction Prefix (Rep)		High	Medium	Set breakpoint on exception handler, Allow single-step/breakpoint exceptions to be automatically passed to the exception handler	High	[56]	1, 2, 3
		Interrupt 3, 0x2D		Low	High				
		Interrupt 0x41		Low	High				
	Debugger Specific	OllyDBG: InputDebugString()		Low	High	Patch entry of kernel32!outputdebugstring()	Low	[19]	1, 2, 3
		SoftICE Interrupt 1		Low	High	Set breakpoint inside kernel32!createfilefilew()			
	Targeted	APT Environment Keying		High	Very High	Exhaustive Enumeration, path exploration techniques	Low	[14, 79]	1, 2, 3, 4
		AI Locksmithing		Very High	Very High	N/A	Rare	[30]	1, 2, 3, 4
Detection-Independent	Control Flow Manipulation	Self Debugging	DebugActiveProcess() DbgUiDebugActiveProcess() NtDebugActiveProcess()	Medium	Low	Set debug port to 0	Low	[131]	1, 2, 3
		Suspend Thread	SuspendThread() NtSuspendThread()	Low	Low	N/A		[54]	1, 2
		Thread Hiding	NtSetInformationThread() ZwSetInformationThread()	Low	Low	Skip the APIs		[138]	1, 2
		Multi-threading	CreateThread()	Medium	Low	Set breakpoint at every entry		[25, 138]	1, 2
	Lockout Evasion	BlockInput(), SwitchDesktop()		Low	Low	Skip APIs	Low	[132, 138]	1, 2, 3, 4
	Fileless (AVT)	Web-based exploits System-level exploits		High	Very High	N/A	Low	[38, 81]	1, 2, 3, 4

Analysis of Methodology

Classification and Comparison of Countermeasure Tactics Against Evasive Malware

Criterion / Countermeasure	Effective Against	Complexity	Weakness	Examples
Reactive	Only Known Evasion techniques	Low	Vulnerable to zero-day techniques.	[18, 69, 133]
Multi-System Execution	Detection-dependent category	Medium	Ineffective against detection-independent tactic.	[6, 61, 58, 63, 74]
Path-Exploration	All tactics except Fileless	High	Vulnerable to anti-symbolic execution obfuscation. Not scalable, resource intensive.	[20, 95, 108]
Towards Perfect Transparency	Fingerprinting Tactic	Very High	Vulnerable to Targeted, Reverse Turing, Stalling, Fileless, and trigger-based tactics. Unless the sandbox is equipped with other countermeasure tactics as well.	[31, 63, 73, 100, 137, 149]

Conclusion

Detection dependant and Detection In-dependant are the two categories for both dynamic analysis mode (manual and automated) of evasion. Reactive approach, Multi-System Execution, Towards Perfect Transparency works best on detection dependant and for detection In-dependant Path-Exploration method with fingerprinting technique will be helpful for detection In-dependant.

References

Anish. (2012, January 7). *Reptile Malware - Behavioral Analysis*. From Malware Analysis: http://malwarecrypt.blogspot.com/2012/01/reptile-malware-behavioral-analysis.html

Awards. (2022, March 1). *AV-TEST*. From AV-TEST Award 2021: The Best in IT Security: https://www.av-test.org/en/news/av-test-award-2021-the-best-in-it-security/

Barlow, J. (2000, March 9). *TFN2k_Analysis-1.3.txt*. From https://packetstormsecurity.com/distributed/TFN2k_Analysis-1.3.txt: https://packetstormsecurity.com/distributed/TFN2k_Analysis-1.3.txt

Cherepanov, A. (2012, June 12). *A new threat for industrial control systems*. From welivesecurity: https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf

Chiu, B. B. (2015, May 4). *Threat Spotlight: Rombertik – Gazing Past the Smoke, Mirrors, and Trapdoors*. From Cisco Blogs: https://blogs.cisco.com/security/talos/rombertik

Hornat, D. D. (2007). *Malware analysis: An introduction. SANS Institute InfoSec.* Reading Room.

Kruegel, C. (2014, March 27). *How To Build An Effective Malware Analysis Sandbox*. From lastline: https://www.lastline.com/labsblog/different-sandboxing-techniques-to-detect-advanced-malware/

Nadji, B. D.-G. (2010, May 7). *See No Evil: Evasions in Honeymonkey Systems*. From Moyix: https://moyix.net/honeymonkey.pdf

Oleg Kulchytskyy, Anton Kukoba. (2021, March 4). *Anti Debugging Protection Techniques with Examples*. From Apriorit: https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software

Oyama, Y. (2018). Trends of anti-analysis operations of malwares observed in API call logs. *Journal of Computer Virology and Hacking Techniques* , 69–85.

Reverse engineering VertexNet malware. (2015, May 8). From cturt: https://cturt.github.io/vertex-net.html

Rozena. (2018, June 29). *Where we go, we don't need files: Analysis of fileless malware*. From gdatasoftware: https://www.gdatasoftware.com/blog/2018/06/30862-fileless-malware-rozena