# LASTLINE WHITEPAPER

## Automated Detection and Mitigation of Execution-Stalling Malicious Code

6950 Hollister Avenue, Suite 101
Goleta, CA 93117
lastline.com | 805.456.7076

## Abstract

Malware continues to remain one of the most important security problems on the Internet today. Whenever an anti-malware solution becomes popular, malware authors typically react promptly and modify their programs to evade defense mechanisms. For example, recently, malware authors have increasingly started to create malicious code that can evade dynamic analysis. One recent form of evasion against dynamic analysis systems is *stalling code.* Stalling code is typically executed before any malicious behavior. The attacker's aim is to delay the execution of the malicious activity long enough so that an automated dynamic analysis system fails to extract the interesting malicious behavior. In a well-known, pioneering scientific paper presented by the researchers *Clemens Kolbitsch, Engin Kirda and Christopher Kruegel* at the ACM Computer and Communications Security Conference in Chicago, IL, in October 2011, the authors presented *HASTEN*, the first tool to detect and mitigate malicious stalling code, and to ensure forward progress within the amount of time allocated for the analysis of a sample. All the of co-authors of this paper, Dr Kolbitsch, Dr. Kirda and Dr. Kruegel, are founding members of Lastline. In this whitepaper, we give an executive summary of that scientific work and use excerpts from the paper.

For a more detailed overview of HASTEN, the reader is referred to the full scientific paper at: http://www.lastline.com/papers/acm_ccs11_hasten.pdf.

Lastline has built novel and improved tools that are conceptually similar to HASTEN, but that are much more efficient and effective in practice. Note that dealing with malware samples that try to evade detection is one of the important leading features of Lastline's analysis technology compared to the competition.

## Introduction

Malicious software (malware) is the driving force behind many security problems on the web. For example, a large fraction of the world's email spam is sent by botnets, Trojan programs steal account credentials for online banking sites, and malware programs participate in click fraud campaigns and distributed denial of service attacks.

Malware research is an arms race. As new anti-malware solutions are introduced, attackers are updating their malicious code to evade analysis and detection. For example, when signature-based anti-virus scanners became widely adopted, attackers started to use code obfuscation and encryption to thwart detection. As a consequence, researchers and security vendors shifted to techniques that focus on the runtime (dynamic) behavior of malware.

An important enabler for behavior-based malware detection are dynamic analysis systems. These systems execute a captured malware program in a controlled environment and record its actions (such as system calls, API calls, and network traffic). Based on the collected information, one can decide on the malicious nature of a program, prioritize manual analysis efforts, and automatically derive models that capture malware behaviors. Such models can then be deployed to protect end-users' machines.

As dynamic analysis systems have become more popular, malware authors have responded by devising techniques to ensure that their programs do not reveal any malicious activity when executed in such an automated analysis environment. Clearly, when malware does not show any unwanted activity during analysis, no detection models can be extracted. For anti-malware researchers, these evasion attempts pose a significant problem in practice.

A common approach to thwart dynamic analysis is to identify the environment in which samples are executed. To this end, a malware program uses checks (so-called red pills) to determine whether it is being executed in a virtual machine or a system emulator such as Qemu. Whenever a malware program is able to detect that it is running inside such an environment, it can simply exit.

Reacting to the evasive checks (red pills), academic researchers have proposed more transparent analysis environments. Another academic approach has focused on the detection of differences between the execution of a program in a virtual platform and on real hardware. When such a discrepancy is identified, the checks responsible for the difference can be removed. Finally, systems that perform multi-path exploration (such as Lastline's analysis environment) can detect and bypass checks that guard malicious activity.

In the next step of the arms race, malware authors have begun to introduce *stalling code* into their malicious programs. This stalling code is executed before any malicious behavior – regardless of the execution environment. The purpose of such evasive code is to delay the execution of malicious activity long enough so that automated analysis systems give up on a sample, incorrectly assuming that the program is non-functional, or does not execute any action of interest. It is important to observe that the problem of stalling code affects *all* analysis systems, even those that are fully transparent. Moreover, stalling code does not have to perform any checks.

With stalling code, attackers exploit two common properties of automated malware analysis systems: First, the time that a system can spend to execute a single sample is limited. Typically, an automated malware analysis system will terminate the analysis of a sample after several minutes. This is because the system has to make a trade-off between the information that can be obtained from a single sample, and the total number of samples that can be analyzed every day. Second, malware authors can craft their code so that the execution takes much longer inside the analysis environment than on an actual victim host.

Thus, even though a sample might stall and not execute any malicious activity in an analysis environment for a long time (many minutes), the delay perceived on the victim host is only a few seconds. This is important because malware authors consider delays on a victim's machine as risky. The reason is that the malicious process is more likely to be detected or terminated by anti-virus software, an attentive user, or a system reboot.

In their work on HASTEN, Dr. Kolbitsch, Dr. Kirda and Dr. Kruegel presented the first approach to detect and evade malicious stalling code, and to ensure forward progress within the amount of time allocated for the analysis of a sample. To this end, they introduced techniques to detect when a malware sample is not making sufficient progress during analysis. When such a situation is encountered, the system automatically examines the sample to identify the code regions that are likely responsible for stalling the execution. For these code regions (and these regions only), costly logging is disabled. When this is not sufficient, the researchers force the execution to take a path that skips (exits) the previously-identified stalling code. At Lastline, we have developed a similar approach to deal with execution-stalling code.

## Problem Description

We define a *stalling code region* as a sequence of instructions that fulfills two properties: First, the sequence of instructions runs considerably slower inside the analysis environment than on a real (native) host. In this context, "considerably slower" means that the slowdown is large compared to the average slowdown that the sandbox incurs for normal, benign programs. Examples of slow operations are system call invocations (because of additional logging overhead) and machine instructions that are particularly costly to emulate (e.g., floating point operations or MMX code).

The second property of stalling code is that its entire execution must take a non-negligible amount of time. Here, non-negligible has to be put into relation with the total time allocated for the automated analysis of a program. For example, for a dynamic analysis system, this could be in the order of a few minutes. Thus, we expect the stalling code to run for at least several seconds. Otherwise, the analysis results would not be significantly affected. That is, when an instruction sequence finishes within a few milliseconds instead of microseconds, we do not consider this as stalling code.

Clearly, an attacker could create stalling code that stalls execution on the real host in the same way it does in the analysis environment. For example, the attacker could use sleep calls, or create high amounts of activity to delay execution. However, in practice, execution delays using sleep-like functions can be easily skipped, and delaying execution (for example, by raising the volume of activity) increases chances of being detected and terminated on a victim host.

Intuitively, our definitions imply that stalling code contains “slow” operations (to satisfy the first property), and that these operations are repeated many times (to satisfy the second property). As a result, attackers typically implement stalling code as loops that contain slow operations (and we will sometimes refer to stalling code as *stalling loops* in this whitepaper).

## System Overview

The goal of our system at Lastline is to detect stalling loops and to mitigate their effect on the analysis results produced by an analysis sandbox. In particular, our system must ensure that the analysis makes sufficient progress during the allocated analysis timeframe so that we can expose as much malicious activity as possible. To this end, our system, just like, HASTEN, can operate in three modes: monitoring, passive, and active mode.

**Monitoring mode.** When the analysis of a malware sample is launched, the system operates in monitoring mode. In this mode, the system performs lightweight observation of all threads of the process under analysis. The goal is to measure the *progress* of each thread, and to identify instances in which the execution might have entered a stalling region.

**Passive mode.** When the monitoring mode detects insufficient progress, this is typically due to slow operations that are executed would succeed, and the malware would terminate without revealing any additional malicious behavior.

To overcome this problem, our approach is as follows: Before we exit a whitelisted code region, we first analyze this region for all variables (memory locations) that the code writes as part of a computation (logic and arithmetic instructions). These memory locations are then marked with a special label (tainted), indicating that their true value is *unknown.* Whenever a machine instruction uses a tainted value as source operand, the destination is tainted as well. For this, we leverage the fact that we have implemented our prototype solution on Lastline’s cloud-based, emulator analysis platform that already supports data flow (taint) tracking.

Whenever a tainted variable is used in a comparison operation or in an indirect memory access (using this variable as part of the address computation), the system temporarily halts the execution of the malware process. It then extracts a backward slice that ends at the comparison instruction and that, when executed, will compute the correct value for this variable. Once the slice is extracted, it is executed on a native machine. As a result, this computation does not incur any overhead compared to the execution on a real victim host.

## Conclusion

As new malware analysis solutions are introduced, attackers react by adapting their malicious code to evade detection and analysis. One recent form of evasion code for dynamic analysis systems is stalling code. In this whitepaper, we present the first academic approach, HASTEN, to detect and mitigate malicious stalling code, and to ensure forward progress within the amount of time allocated for the analysis of a sample. We have reimplemented HASTEN within Lastline, and our results show that the system works well in practice, and is able to reveal additional behaviors in real-world malware samples that contain stalling code that are invisible to competing products in the market.

## About Lastline

Lastline is a technology pioneer dedicated to stopping advanced malware, zero-day attacks, drive-by downloads and sophisticated Advanced Persistent Threats. Lastline's flexible Previct platform provides high-resolution analysis and protection; the required network security foundational layer capable of providing exacting security legacy APT, IPS, AV and next generation firewalls simply cannot see. The Santa Barbara based company is dedicated to providing the most accurate malware detection and defense available to our customers.