A Testbed for Malicious Code Detection: A Synthesis of Static and Dynamic Analysis Techniques*

R. Crawford, R. Lo, J. Crossley, G. Fink, P. Kerchen, W. Ho, K. Levitt, R. Olsson, and M. Archer

Division of Computer Science
University of California, Davis
Davis, CA 95616

Abstract

This paper proposes an environment for detecting many types of malicious code, including computer viruses, Trojan horses, and time/logic bombs. This Malicious Code Testbed (MCT) is based upon both static and dynamic analysis tools developed at the University of California, Davis, that have been shown to be effective against certain types of malicious code. The testbed enhances the power of these tools by using them in a complementary fashion to detect more general cases of malicious code. The MCT allows administrators and security analysts to check a program before installation, thereby avoiding any damage a malicious program might inflict.

Keywords: Detection of Malicious Code, Static Analysis, Dynamic Analysis.

1 Introduction

In the past five years, there has been an explosion in the number of Trojan horses, time bombs, and viruses that have been found on computers. Furthermore, the ease with which one may write a virus or trapdoor is certainly cause for concern: In his Turing Award lecture, Ken Thompson demonstrated an elegant yet simple trapdoor program that was quite effective in subverting the security of a UNIX system [5]. The situation is even less encouraging in the personal computer arena — literally hundreds

*SPONSORS: National Computer Security Center, Lawrence Livermore National Laboratory, Deloitte Touche

CS-1011
Cisco Systems, Inc. v. Finjan, Inc.

4.2.1 Disassembly — Translation into Internal Form

To statically analyze the behavior of an executable machine code program, we must first "disassemble" it, that is, translate its code into our internal form. We have designed a set of procedures that, given a 2-tuple *(Memory_Address, Memory_Contents)*, will disassemble its *Memory_Contents*,

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

of computer viruses, time bombs, and Trojan horses exist for all of the major personal computers in use today.

Malicious code detection — in the most general case — is known to be an undecidable problem. However, a number of techniques already exist for coping with certain restricted forms of malicious code. Although no algorithm that identifies malicious code in all its guises can exist, various heuristics may be applied to detect, e.g., viruses that are not very cleverly hidden in legitimate code. Such an approach to managing the problem is valid toward all forms of malicious code: stopping a large percentage of destructive programs is a considerable improvement over not stopping any of them.

This idea forms the basis for a *Malicious Code Testbed* (MCT) capable of detecting a large majority of current and future malicious programs. Given the absence of a decision procedure for *malicious* code, such a testbed would allow one to examine a program to ascertain whether or not it is *suspicious*. The MCT will employ heuristics to analyze and simplify a program. Although these techniques may not always succeed, they should cover all "tricks" thought to be employed by malicious code.

We first discuss some of the known methods of coping with malicious code, and then summarize previous work at UC Davis aimed at providing defenses against malicious code. Finally, we explore in greater detail the idea of the *Malicious Code Testbed* that builds on our previous work, and melds several different heuristic techniques into a more effective, integrated system.

2 A Sample of Current Methods for Coping with Malicious Code

Presently, the majority of malicious code defenses are concerned with computer viruses. However, some are more broadly applicable to malicious code in general. Table 1 shows the applicability of some of these methods. These methods can be divided into two distinct classes depending on when they are applied: as a *pre-execution* check or at *run time*. Pre-execution techniques are applied to a suspicious program before it can be executed by a user. In contrast, run time methods are actually applied to the program as it executes, in hopes of stopping the program before it can cause damage or allow a virus to propagate.

Many of the more sophisticated pre-execution methods rely on the prior existence of a copy of the program that is assumed to be "clean", perhaps because it was originally written by a trusted programmer and then translated into an executable file by a trusted compiler on a secure system. One such method computes cryptographic checksums that are characteristic of that trusted executable file, and embeds them in that file. The file is then copied to an insecure environment, whose operating system will not allow a user to execute any program until it has recomputed what those checksums should be and compared those values with the ones

stored in a table in the MCT.

4.2.3 Memory Model for the Data Segment

Cells in a suspicious program's data memory can be represented by the same structures as are used for its code, although at first it might appear that only the *(Memory_Address, Memory_Contents)* fields are needed.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

actually embedded in the program. In this way, most alterations made to a trusted executable file after it leaves the secure system can be detected before the program is executed in the insecure environment.

It is important to note that this technique shares one important characteristic in common with most other sophisticated pre-execution methods — ultimately, they depend on the prior application of detection (or formal verification) techniques in order to certify an executable file as "trusted" in the first place.

	PACL	Static Analyzer	Simple Scanner	Run-time Monitor	Encryption	Watchdog Processors	Dynamic Analyzer
Covert Channel	none	low	none	limited	high	none	high
Worm	high	low	none	low	none	none	low
Trojan Horse	high	high	low	high	low	none	high
Virus	high	high	low	high	high	high	high
Time Bomb	high	high	low	high	low	none	high
Trapdoor	none	high	none	none	low	none	high
Salami	none	low	none	none	none	none	high

Table 1. Applicability of Defenses.

2.1 Simple Scanners and Monitors

Simple scanners such as McAfee's Scanv or Norstad's Disinfectant are by and large the most common pre-execution method in use today. Typically, the user will invoke a scanner to search a binary program for patterns (bitstrings) that match those of known malicious programs. If none of those bitstrings are found, the user then proceeds to execute the program. Thus these scanners boast a very good record in defending against *known* malicious programs but they cannot be applied in general to finding new or mutated malicious code. Another popular approach uses *simple monitors* to observe program execution and detect malicious behavior at run time. Such monitors usually watch all disk accesses to ensure that no unauthorized writes are made. Unfortunately such techniques incur a substantial speed penalty during execution. In addition, to be effective these programs must err on the conservative side, resulting in many false alarms which require user interaction.

4.3.1 Distinguishing "Code" from "Data"

Contrary to popular belief, disassembly can be a formidable task. In general, it is formally undecidable to determine whether a region of "data" could be executed as code (the so-called "state-entry problem" for Turing machines), or to determine whether a region of "code" could ever be

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

2.2 Program Access Control Lists

The next approach, *program access control lists* (PACL's) [6], consists of associating with each file in a system an access control list that specifies what *programs* can modify the file. This preventive approach has the effect of limiting damage that can be done by many malicious programs, especially Trojan Horses unknowingly executed by the user. In contrast, conventional ACL's would be ineffective in such situations because they only limit which *users* can modify the file. Although the PACL approach can help to ensure integrity, it is ineffective against attacks exploiting covert channels that only violate information security, not integrity.

2.3 Encryption & Watchdog Processors

Encryption is another method of coping with the threat of malicious code. Lapid, Ahituv, and Neumann [2] use encryption to defend against Trojan horses and trapdoors. When correctly implemented, encryption techniques are quite effective against many types of malicious code, but the cost of such a system is high due to the required hardware. Similarly, *watchdog processors* [3] also require additional hardware. Such processors are capable of detecting invalid reads/writes from/to memory, but they require additional support to effectively combat viruses. Also, both of these methods are dependent on the prior existence of a "clean" version of every program that is to be executed. As mentioned, to certify such copies as "clean" in the first place requires either formal verification or a malicious code detection capability, which is the subject of the present paper.

3 Ongoing Work at UC Davis

We are pursuing two families of analytical techniques that, unlike most current virus prevention and detection methods, attempt to dissect a program to determine what it does and how. By examining the code, *static analysis* techniques can determine certain properties for some types of programs. *Dynamic analysis* techniques attempt to learn about a program's behavior by actually running it in a "cleanroom" environment or by simulating its execution.

At UC Davis, three analysis tools have been developed to help in determining whether a program has any potentially malicious code in it: VF1, Snitch, and Dalek. VF1 uses data flow algorithms during static analysis to determine the names of all files that a program can access. Snitch statically examines a program for duplication of operating system services. Dalek is a debugger that forms the basis for a dynamic analyzer.

3.1 Static Analysis

From Table 1, one can see that *static analysis* [1] can be applied to a broad class of security problems. By closely examining the binary or source code

data. As with many programming languages, this concept of "failure" may need to propagate outward (to other *Memory_Addresses*), perhaps even requiring certain aspects of the static analysis to be re-computed from scratch the next time the interpreter needs to rely on a field computed by static analysis.

3. Invoke the translator routines to disassemble the *Memory_Contents* and record the resulting internal form syntax in the appropriate

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

of a program, static analysis attempts to detect the presence of malicious sections in that program. However, in the most general case such detection is known to be an undecidable problem, resulting in a need for more selective analysis techniques aimed at limited subclasses of problem instances. A generic virus designed to infect an *arbitrary* program will be unlikely to evade detection. In contrast, malicious code intended for one specific program can be more smoothly integrated with other sections of that program, thereby effectively camouflaging its presence. In such cases, it is more cost-effective for the detection to focus on the services provided by the operating system that might be exploited by the malicious code, and on the strategic vulnerabilities of that operating system and underlying architecture. Thus our approach avoids the prohibitive cost of formal program verification in favor of slicing [1] and other static and dynamic analysis methods that reduce the problem space to a more tractable size.

3.1.1 VF1

VF1 is a prototype system that was implemented to determine the viability of applying static analysis to the detection of malicious code; it uses a technique called *slicing*. Slicing involves isolating the portions of a program related to a particular property in which one is interested. The sliced program (which is greatly reduced in size compared to the original) can then be analyzed to give information about that particular property. VF1's target property is filename generation — in particular, which files can be opened and written to by a given program. By knowing what files a program can write to, one can determine if there is a possibility of the program being a virus. For example, if a program that does not need to write to files (e.g., *ls*, the UNIX directory listing program) possesses code to open and write any file, then one might suspect that the program contains a virus.

VF1 translates a program written in the C programming language to a program expressed in a Lisp-like internal form that is easier to analyze This resultant program can then be sliced with respect to any given line of its code. That is, one can select a line of the resultant program that performs an interesting action (such as opening a file for writing), and have VF1 determine which statements of the resultant program have bearing on that selected line.

3.1.2 Snitch

Snitch is a prototype that detects duplication of operating system access routines in a program. Its strategy relies on the fact that most UNIX programs contain at most one instance of any operating system service (e.g., open, write, close). Since a simple virus cannot rely on all programs possessing the services it needs, it usually carries each of these services with it, inserting them into every program it infects. This will most likely result in a duplication of some operating system services. When Snitch is used to analyze an infected program, it will report this duplication

behavior would be grounds for more extensive dynamic analysis. How could dynamic analysis detect self-modifying code? Primarily for expository purposes, we first introduce a *special case* of self-modifying code that builds on the two examples of the last section. Later, we describe self-modifying code in a more general context.

Suppose that in the second example of the previous section, the *Mem-*

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.