Automated Assistance for Detecting Malicious Code *

R. Crawford, P. Kerchen, K. Levitt, R. Olsson, M. Archer, M. Casillas

Department of Computer Science
University of California, Davis
Davis, CA 95616
Email: virus@cs.ucdavis.edu

Abstract

This paper gives an update on our continuing work on the Malicious Code Testbed (MCT). The MCT is a semi-automated tool, operating in a simulated, cleanroom environment, that is capable of detecting many types of malicious code, such as viruses, Trojan horses, and time/logic bombs. The MCT allows 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

The Malicious Code Testbed (MCT) was originally designed to use 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. One goal of the testbed is to enhance the power of similar tools by using them in a complementary fashion to detect more general cases of malicious code.

In our report to this conference last year [1], we presented a design overview of the MCT. In the present paper, we report on our progress towards upgrading the MCT environment for dynamic analysis.

Although, in principle, the notion of a Malicious Code Testbed is independent of any particular operating system or architectural platform, our initial implementation efforts have focused on simulating a DOS operating system running on PC architectures. This design decision was made primarily because the PC/DOS environment is so widespread and accessible to intrusions; thus this environment is the one that has engendered the most real-world malicious code we can use to challenge our detection techniques.

Sections 2 and 3 provide background material on malicious code and current detection methods. Section 4 reviews the use of events in dynamic analysis techniques, and Section 5 describes the architecture of the MCT. Section 6 presents some results from our experience using the MCT on malicious code.

*SPONSORS: Lawrence Livermore National Laboratory, U.S. Department of Energy

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

2 Malicious Code — A Brief Overview

In recent years, various forms of malicious code have appeared on virtually all major families of computer platform. The prevalence of malicious code — Trojan horses, time bombs, worms, and viruses — threatens the traditional "open systems" approach that has evolved in the academic realm, as well as in much of the commercial sector.

The current situation in the personal computer arena may be indicative of future trends in workstation and mainframe environments. On PC systems — where literally hundreds of computer viruses, time bombs, and Trojan horses have proliferated — the problem is caused by rogue programs that unwittingly are *invited in* to the system. Thus malicious code may be inserted into almost *any* type of computer system via these same avenues — "shareware" may be installed, or malicious code might be produced in-house by a disgruntled employee, or a program containing malicious code might even be purchased from a legitimate vendor of commercial software.

Our definition of what constitutes "malicious" code shall address only the probable *effects* of executing such code; we shall not concern ourselves with the "original intent" of the (possibly unknown) writer. Although the intentions of the writer may be crucial in determining legal culpability — e.g., whether malice and forethought were present — to include such considerations within the scope of our "working definition" for malicious code would clearly render the problem incomputable.

Yet even using our restricted, *operational definition* of "malicious code", the problem of malicious code detection — in the most general case — is not decidable by purely formal methods. This follows not merely from the results of [4] [2] [3], but rather because the inherent semantics of the problem statement demand that a *value judgement* regarding the nature of the code's probable effects be rendered. But because doing so would require that the *intent* of the program's potential *users* be considered, no article of faith akin to Church's Thesis can serve to bridge the gap between our *intuitive* sense of "malicious effects", and *algorithmic* solutions. It would seem that, in all but the most severely restricted programming environments, the problem statement must remain a fuzzy one.

Thus, although no algorithm that identifies malicious code in all environments and in all guises can exist, a number of techniques already exist for coping with certain restricted forms of malicious code. Since the problem cannot with certainty be *prevented* in current programming environments, it must be *managed* instead.

This idea forms the basis of the Malicious Code Testbed — an automated assistant whose mission it is to perform the "grunt work" necessary to aid a human analyst in detecting not only currently known forms of malicious code, but also mutated or entirely novel forms. 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*.

We first discuss the most prevalent methods of coping with malicious code, and then describe some of our previous work aimed at providing defenses against malicious code. Then we explore in greater detail the *Mclicious Code Testbed*.

3 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. These methods may 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. Another taxonomy of malicious code defenses divides all methods into the categories of *static* or *dynamic* analysis. Although most static analysis techniques are applied as pre-execution checks, certain static analysis techniques can be applied at run time. Similarly, although most dynamic analysis techniques are applied as run time checks, certain dynamic analysis techniques (such as our own Malicious Code Testbed) can be applied as pre-execution checks.

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. [6] 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 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 "clean" in the first place.

Keeping Ken Thompson's admonition "on trusting trust" firmly in mind [5], how should a security administrator proceed when faced with programs so large or complex that "trust, but verify" is not a feasible option? We suggest that — in the middle ground between the two extremes of exhaustively provable correctness and trust based on nothing more substantial than personal familiarity with, or a background security check on, a program's writer — the MCT (acting to assist a human analyst) can provide a practical alternative basis for trust.

3.1 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 the static text of a binary program for fixed 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, such as polymorphic viruses that use a *known* "Mutation Engine", but they cannot be applied in general to finding new malicious code, or even to finding familiar malicious code protected by a "Mutation Engine" that is, itself, slightly mutated. Another popular approach uses *simple monitors* to observe program execution and detect potentially malicious behavior at run time. Such monitors usually sit astride the system call interface, e.g., to watch all disk accesses and ensure that no unauthorized writes are performed. Unfortunately such techniques incur a substantial speed penalty during execution of normal programs, and typically become quite a nuisance to the user.

To be effective, these programs must also err on the conservative side, resulting in many false alarms which require user interaction. But ir these interactions, current techniques require the user to make relatively immediate (and usually uninformed) decisions regarding whether the program should be allowed to pro-eed. Such decisions would benefit immensely from the opportunity to explore a trace of the program's history, as well as its then-current execution state.

3.2 Encryption & Watchdog Processors

Encryption is another method of coping with the threat of malicious code. Lapid, Ahituv, and Neumann [7] 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* [8] 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.

4 Review of Dynamic Analysis using Events

Over the last few years, we have developed a powerful, state-of-the-art debugger called Dalek [9]. Dalek incorporates two significant advances over traditional debuggers: it features a *fully-programmable language* for manipulating the debugging environment, and it provides extensive support for user-definable *events*.

The MCT user's environment was designed in accordance with the philosophy underlying the Dalek debugger, and features analogous to those in Dalek have been incorporated into the MCT. But we have also customized the MCT environment, in light of its specific mission to help ferret out malicious code. We believe that "dynamic analysis" (and the development of appropriate methodologies for it) should be seen as representing an extremely promising avenue of inquiry, rather than as being just a fancy word for the sorts of things people have always done with traditional debuggers.

By *fully programmable*, we mean the MCT is an *extendible environment*, in a similar sense that the Emacs text-editor is extendible. But due to the nature of the MCT's mission, these general-purpose language constructs have been fully integrated with traditional application-specific debugging features such as breakpoints and single-stepping.

Like the Dalek debugger, the MCT also provides automated support for detecting hierarchical *events* — occurrences of interesting activities during the execution of the suspicious program. This capability allows the MCT to represent the suspicious program's behavior in terms of whatever higher-level abstractions have been defined by the security analyst.

In some ways, an event is conceptually similar to a tuple in a relational database — once the structure of a particular database table has been defined by the user, every *occurrence* of an event of that type that is detected by the MCT will have its *attributes* recorded permanently, as fields in a newly inserted tuple. That is, when the MCT detects an event occurrence, it causes a corresponding tuple (or record) to appear in the appropriate database table. The attributes associated with an event should contain information sufficient to characterize a particular occurrence of that event, allowing it to be distinguished from other instances of the same event. The code written by a security analyst for an event's definition can cause it, upon activation, to assign values to these attributes from variables in the suspicious program, from variables in the "outer" MCT environment, or from computation based on a combination of such variables.

In addition to defining an event as a template for passive data, the security analyst also needs to define an active, procedural aspect for that event. This is accomplished by writing a body of code in the MCT's language, and associating it with that event. The purpose of this code, when activated, is to recognize exactly those conditions in the suspicious program's execution state that the security analyst has specified as constituting a valid occurrence of this particular type of event.

This event-recognition code can be executed manually by the security analyst as s/he single-steps the suspicious program, or it can be executed automatically by the MCT, if the analyst has bound that event's code to a breakpoint, or to a *range* of breakpoint addresses. Events whose code is activated in this manner are called *primitive* events.

The MCT also supports *high-level* events. When defining a high-level event, one must specify the names of all lower-level events on which it *depends*. A high-level event is not explicitly raised; instead, the MCT can automatically trigger a high-level event's code into executing whenever an occurrence of a primitive event on which that high-level event *depends* is successfully recognized. The high-level event's code will have access to all the attributes of its lower-level constituent events, as well as access to the "raw" state of the suspicious program and to variables defined in the "outer" MCT environment.

Note that the security analyst can define a high-level event whose recognition may depend on lower-level constituent events whose occurrences are *widely separated in time*. For a concrete example of a network of events used to detect self-propagating code, see [1].

Viewed from the perspective of a relational database, a high-level event is conceptually akin to an ongoing query: In defining a high-level event, the security analyst poses a query. The MCT then provides incremental answers to that activated query, as the behavior of the suspicious program causes new occurrences of primitive event/attribute tuples automatically to be inserted in the database.

The "execution history database" maintains a record of all recognized event occurrences and their attributes. It may be browsed selectively by the security analyst in interactive mode, or accessed programmatically via access functions written in the MCT's language.

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.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase®
Smarter legal research.