UCRL-JC-105792
PREPRINT

Towards a Testbed for Malicious Code Detection

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

Division of Computer Science
University of California, Davis

Compcon '91
San Francisco, CA

February 25 - March 1, 1991

Lawrence Livermore National Laboratory

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

UCRL-JC--105792

DE91 007632

Towards a Testbed for Malicious Code Detection *

R. Lo, P. Kerchen, R. Crawford, W. Ho, J. Crossley, G. Fink, 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, which have been shown to be effective against certain types of malicious code. The testbed extends the usefulness of these tools by using them in a complementary fashion to detect more general cases of malicious code. Perhaps more importantly, 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 a simple trapdoor program which was quite effective in subverting the security of a UNIX system. The situation is even less encouraging in the personal computer arena: literally hundreds of computer viruses, time bombs, and Trojan horses exist for all of the major personal computers in use today.

However, there are techniques for coping with these problems. While one will never be able to distinguish a cleverly disguised virus from legitimate code, one may detect a not-so-cleverly hidden one. The same holds true for all malicious code: stopping a large percentage of destructive programs is considerably better than not stopping any. This idea forms the basis for a *malicious code testbed* (MCT) capable of detecting a large majority of current and future malicious programs. Such a testbed would allow one to examine a program to ascertain if it is suspicious. In the following section, we present a taxonomy of malicious code with examples. Following the taxonomy, we discuss many of the known methods of coping with malicious code. We then summarize the progress which has been made at UC Davis. Finally, we propose the idea of the *malicious code testbed*, which combines this previous work into a more effective system.

2 Taxonomy of Malicious Code

Computer security should insure that no unauthorized actions are carried out on a computer system. Security is violated when someone succeeds in retrieving data without authorization, destroying or altering data belonging to others, or locking up computer resources to make them unavailable. Malicious programs are those programs which cause such violations.

To categorize malicious activities, we may examine the following aspects of a malicious program [Table 1].

What are the malicious actions?

A malicious program may not only directly retrieve or alter confidential information, but it may also modify the security state of the computer system so that an unauthorized person could access this information. Therefore, malicious activities refer to all activities leading to such consequences.

How do malicious programs obtain privilege?

Before any damage can be done, the malicious program must obtain the required privilege from an authorized user or from the operating system. A common way is to act as a Trojan horse, claiming to perform some useful functions, but performing others in addition or instead. A malicious program can also obtain privilege from the operating system by exploiting system bugs, taking advantage of administrative flaws, or faking authentications.

* Sponsored by Lawrence Livermore National Laboratory

	Malicious Actions	Obtain Privilege	Distribution Channels	Triggers
Covert channel	Disclose information	Installed by programmers		Useful information found
Worm	Exhaust resources, Any	Writer starts it, self-replication	Network	
Trojan horse	Any	User execution	Exchange of software	User execution
Virus	Infect programs, Any	User execution, self-replication	Contact with an infected system	Execution
Time/Logic Bomb	Any	Installed by programmers		Time/date, conditions satisfied
Trapdoor	Gain privilege	Holes implanted by programmers		Started by attackers
Salami Attack	Embezzlement	Installed by programmers		Execution

Table 1. Types of Malicious Code

How do malicious programs enter a system? Sometimes a malicious program is advertised as public domain software available in public bulletin boards; security may be compromised if any user copies and executes such programs on his computer. Another similar example is that of the Christmas Virus, which replicates by sending copies of itself to users and requesting them to execute the message. In cases of planned attacks, trapdoors previously implanted in the system are used by the malicious programs.

How are the malicious actions triggered?

A malicious program may stay dormant for an indefinite period. It works normally until a scheduled moment or certain conditions are satisfied. For example, a malicious program which exploits covert channels may only be active when confidential information is being displayed on a terminal; at other times, it may sleep or perform some diversionary action.

3 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 2 shows the applicability of some of these methods. One can classify these methods into two classes: *preventive* and *detective*. While prevention is important, detection is preferable since it does not rely on a program being in a "clean" state. Thus, detective approaches appear to be more generally applicable.

3.1 Program Access Control Lists

The first approach, *program access control lists* (PACL's) [5], 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, but it is ineffective against attacks such as covert channels which only violate information security, not integrity.

3.2 Static Analyzers

From Table 2, one can see that *static analysis* [1] can be applied to a broad class of problems. By closely examining the binary or source code of a program, static analysis attempts to detect the presence of suspicious sections in that program. However, in the most general case such detection is incomputable, resulting in a need for more selective analysis techniques. Since malicious code in general can be more smoothly integrated with the code of the program it is infecting, detection must be focussed on the strategic vulnerabilities of the operating system and underlying architecture in question. In this way, more generalized detection is possible without the full cost of program verification because slicing [1] and other static and dynamic analysis tools will reduce the problem space to a tractable size.

3.3 Simple Scanners & Monitors

Simple scanners are by and large the most common means of malicious code detection in use today. Typ-

	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 2. Applicability of Defenses.

ically, a scanner will search a program for patterns which match those of known malicious programs. As a result, these programs 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. Such monitors usually watch all disk accesses to insure that no unauthorized writes are made. Unfortunately, for these programs to be effective, they must err on the conservative side, resulting in many false alarms which require user interaction.

3.4 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, trapdoors, and other problems. When correctly implemented, such a system would be 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 would require additional support to effectively combat viruses. Also, both of these methods are preventive in that they require a "clean" version of every program which is to be examined. In many instances, such clean copies are not available, thereby limiting the usefulness of these approaches.

3.5 Dynamic Analyzers

Finally, *dynamic analysis* offers a reasonable potential for detection of a large class of malicious code. By observing a program at run-time in a controlled environment, one can determine exactly what it is trying to do. However, like static analysis, this technique must be used "off-line" to allow the analyzer to keep track of the program's actions. As a result, clever programs can elude the analyzer by only executing when they "know" that they are not being watched.

Unlike most virus detection techniques, two types of analysis attempt to peer inside a program to detect what it is doing and how. Static analysis methods can determine certain properties for some types of programs. Dynamic analysis methods attempt to learn more about a program's behavior by actually running it or by simulating its execution.

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

4 Static Analysis Tools

4.1 VF1

VF1 is a prototype system that has been 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 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 be suspicious that the program contains a virus.

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

4.2 Snitch

Snitch is a prototype of a *detector* of duplication of operating system calls. This detector makes use of 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 will carry all of those services with it, inserting them into every program it infects. This will most likely result in a duplication of some OS services. When Snitch is used to analyze the infected program, it will report this duplication as being suspicious. The Snitch prototype is specific to Sun-3's running SunOS, but many of the concepts underlying the prototype can be applied to other architectures and operating systems.

Snitch consists of two major modules. The first module, the disassembler, takes an executable program as input and produces the equivalent Motorola 68020 assembly language representation as output. The second module, the analyzer, takes the output from the disassembler and examines it for duplication of OS services, reporting any such duplications as well as the number of occurrences of all system calls.

5 Debugger-based Dynamic Analysis

One obvious approach to dynamic analysis is to base the analysis on a debugger. Over the last two years, a debugger called Dalek has been developed at UC Davis [4]. Dalek offers support for the notion of user-definable *events*. The user defines an event template by writing Dalek language code (e.g., employing IF or WHILE statements) that will be executed by Dalek as it attempts to recognize different occurrences of that event. One typical form of primitive event might be defined to capture certain details of a procedure's invocations, e.g., the values of its actual parameters. Another typical form of primitive event might be defined to capture the value of a particular variable every time it changes.

Hierarchical events can also be defined. High-level events are used to correlate and combine (e.g., through Dalek's IF or WHILE statements) the attributes from instances of two or more primitive events that may have occurred widely separated in time. In this way, the user can construct behavioral abstractions - models or patterns that characterize the activity of the application program.

One can imagine how such capabilities might be applied to the detection and understanding of viruses or other malicious code but it might seem that in real-world situations, such event-based methods would be ineffective against hostile or secretive programs. In the first place, one would expect that the malicious code would have been stripped of all (correct) symbolic information. Thus the debugger would not know the names, sizes, or locations of procedures or data structures. However, most operating systems offer some assistance in this regard, allowing a relatively complete behavioral trace of all system-related activity initiated by a suspicious program to be obtained. Secondly, a malicious program may alter its own code, making analysis difficult. Under Dalek, however, one may define events to recognize such self-modifying behavior. Therefore, self-modification does not present insurmountable difficulties for the debugger but it does increase its complexity.

Figure 1 illustrates how high-level events can be used to correlate attributes captured by lower-level events to provide a characterization of a suspicious program's behavior represented in terms of whatever semantic models the user has determined are most relevant.

We envisage equipping Dalek with a library of predefined events to capture suspicious and malicious behavior, similar in spirit to the events shown in Figure 1. For example, attempting to open (or change/inspect the permissions on) all files in the current directory might be considered suspicious. Writing the same block of "data" to several different executable files would appear even more suspicious.

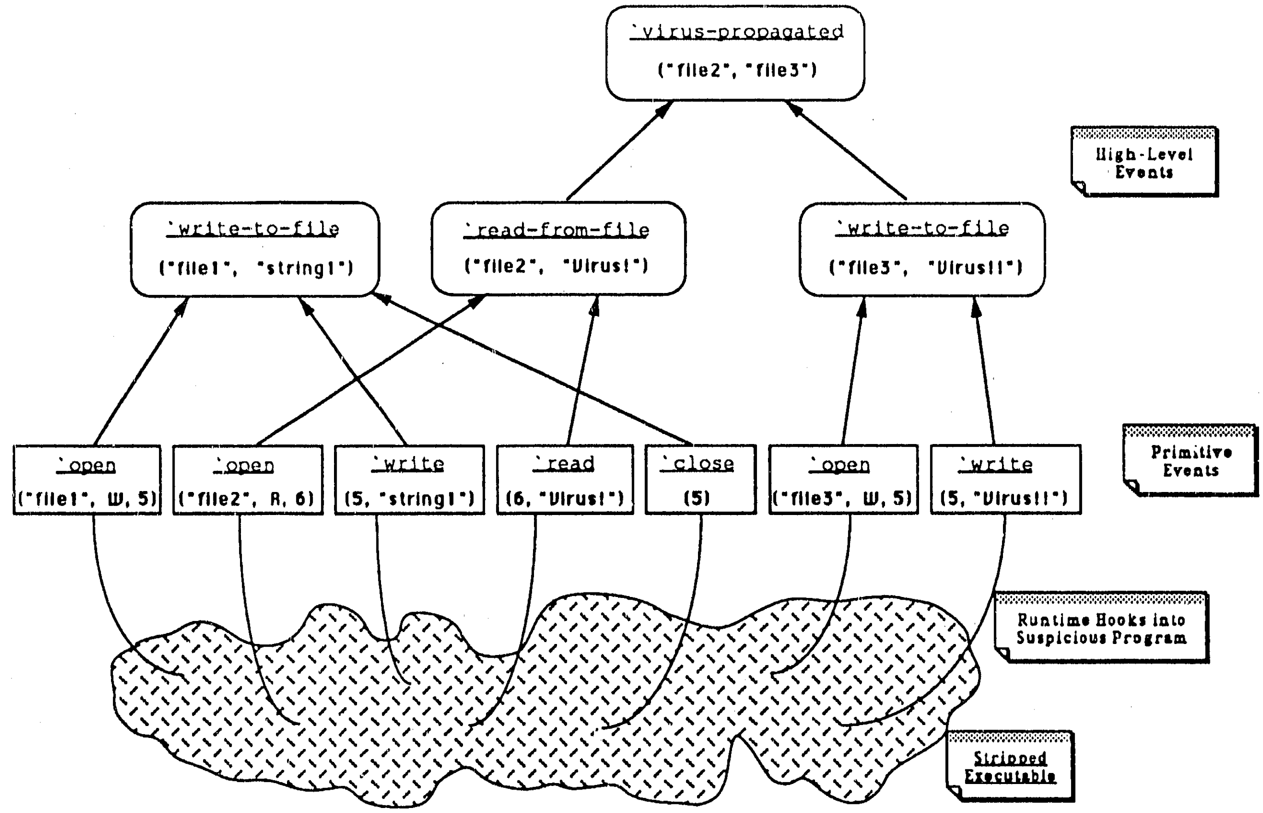

Figure 1. Interaction of Events in Dalek

6 Towards a Testbed

The *malicious code testbed* (MCT) under development consists of a set of tools that will assist a user in detecting viruses and Trojan horses and in identifying programs which exploit security flaws within developed software. It is based in part on the three tools mentioned above: Dalek, VF1, and Snitch.

The primary goal is to provide an environment and tools to assist in the identification of malicious logic in developed software. Since malicious code detection is an incomputable problem, the tools will not be able to give a yes-no answer. Instead, the software is analyzed and its properties summarized to allow the analyst to understand the effect of its execution. The tools will identify suspicious code but it is up to the user to make the final decision about whether or not the code is malicious. For example, our tool may indicate that a program would destroy all information in the current directory. Most people would consider this a malicious activity. However, the program is not malicious if the intention of the user is to clean up his directory by using such a program.

The other goal is to further examine a suspicious program identified by the MCT. The purpose of this further examination is to determine the severity of the identified suspicious activity, locate other suspicious activities, determine its triggering conditions, and produce signatures that may be used to locate the existence of identical or similar malicious logic in other programs.

The MCT employs two kinds of analysis techniques: *static analysis* and *run-time*, or *dynamic*, *analysis*. Both techniques are necessary because they are applied in different situations, thus complementing each other. Compared with static analysis, dynamic analysis is less computation intensive and able to follow any execution sequence even if the program modifies itself on the fly. However, since only some executed sequences are tested, dynamic analysis can certify only the existence of certain activities, i.e. violation of security policy, but it cannot indicate their non-existence. Therefore, both are needed.

Architecture

The static analysis tool works in 5 stages: processor-dependent disassembly, intelligent decompilation, data flow analysis, slicing, and symbolic simplification. The processor-dependent disassembly stage translates an executable program into an intermediate form, and then the decompiler attempts to determine how variables are allocated in the program. Knowing where the variables are stored, the data flow analyzer determines the relationship between variables, i.e. which variables influence the value stored in designated variables. Slicing produces a bona-fide program that computes the value of the variables in question. Finally, the symbolic simplifier tries to simplify the bona-fide program as much as possible.

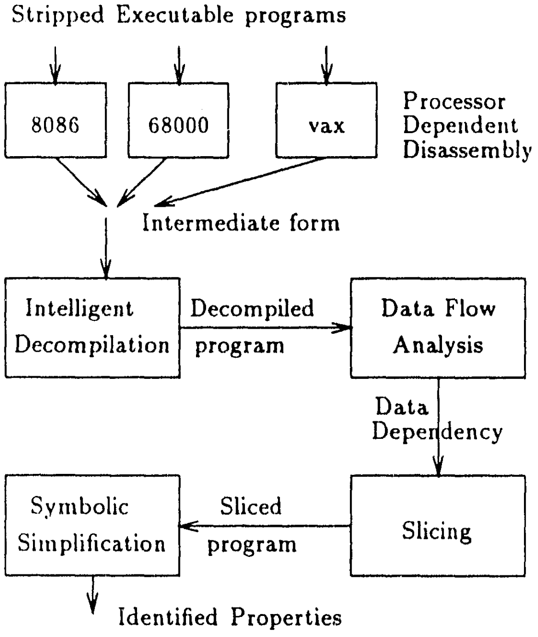

Figure 2. Architecture of the static analyzer.

6.1 Complementary Use of Static and Dynamic Analysis

After some degree of intelligent decompilation, the static analyzer attempts to slice the program to identify all sections of the code involved in the generation of filenames.

Static analysis also needs to look for other discernible suspicious properties. Suppose, for example, that the code resultant from slicing with respect to filename generation is not reachable from the main entry point of the original program. This indicates a very poorly written program or a program that modified itself in order to reach the sliced section. Similarly, any other indications of self-modifying behavior would be grounds for more extensive dynamic analysis.

Dynamic analysis can force the suspicious program to execute certain sections, suspending it periodically to communicate its status to the static analyzer. During this phase, the user of the MCT might devise various hypotheses explaining the goals of the suspicious program and explaining its methods in pursuit of those objectives. For example, a program which encrypts part of its code would be analyzed dynamically in order to examine the decrypted code. This iterative process could continue until the MCT user had gained a thorough understanding of the goals and methods of the suspicious program.

An Example

The ftp program associated with Sun Unix 3.4 has a bug that allows a masquerader to login as any other user provided that he has successfully logged into the system once. The masquerader first logs in with a valid userid and password, causing the *logged_in* variable to be set to 1. Then s/he performs a login with a 'victim' userid and any password, exploiting the flaw that the *logged_in* variable is not reset. Since *check_login* only checks *logged_in*, the 'victim' userid is assumed to be logged in the system. The essential stripped code, written in pseudocode, is as follows:

```
ftp:  USER username CR
          { set new user id
                ... no reset of the variable logged_in }
      | PASS password CR
          { if password is correct, set logged_in to 1
      else print error message }
check_login:
      { valid_login = logged_in; }
```

This bug can be identified with static techniques in two ways. The first method is to check for a data flow anomaly in the data dependency graph. We can see that the variable *logged_in* is never initialized in the program. The second method is to expand *logged_in* symbolically to see how it is computed. The symbolic output will indicate that its value is set with a correct password, but not changed with an incorrect password.

7 Conclusions and Future Work

We have described a testbed under development which detects malicious code that other techniques cannot detect. This testbed uses static and dynamic analysis techniques in a complementary fashion to identify suspicious programs before they are installed and allowed to cause any damage. The static analysis uses slicing to reduce a program to a size which allows verification techniques to discover any suspicious code. The dynamic analysis uses an event-based debugger which is capable of analyzing code which static analysis cannot. We will be applying this testbed on known instances of malicious code, especially viruses, worms, and the like. Future work will include formalizing the concepts of *maliciousness* and *suspiciousness* and improving the static analysis techniques used to discover the meanings of loops in sliced programs.

Acknowledgements

We thank Doug Mansur and Bill Arbaugh for their valuable insights.

References

[1] P. Kerchen, R. Lo, J. Crossley, G. Elkinbard, K. Levitt, and R. Olsson. "Static Analysis Virus Detection Tools for UNIX Systems", *Proc. of NIST/NCSC 13th Nat'l Computer Security Conf.*, Washington, DC, Oct. 1-4, 1990, pp. 350-365.

[2] Y. Lapid, N. Ahituv, and S. Neumann. "Approaches to Handling 'Trojan Horse' Threats", *Computers & Security*, Vol. 5, 1986, pp. 251-256.

[3] A. Mahmood and E. J. McCluskey. "Concurrent Error Detection Using Watchdog Processors–A Survey" *IEEE Transactions on Computers*, Vol. 37, No. 2, 1988, pp. 160-174.

[4] R. Olsson, R. Crawford, and W. Ho. "A Dataflow Approach to Event-based Debugging", to appear in *SOFTWARE-Practice and Experience.*

[5] D. Wichers, D. Cook, R. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo. "PACL's: An Access Control List Approach to Anti-Viral Security", *Proc. of NIST/NCSC 13th Nat'l Computer Security Conf.*, Washington, DC, Oct. 1-4, 1990, pp. 340-349.

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

END

DATE FILMED

03 / 01 / 91