Trusting Software: Malicious Code Analyses

Sam Nitzberg, et. al.

Abstract – **Malicious code is a real danger to defense systems, regardless of whether it is a programming flaw that can be exploited by an attacker, or something more directly sinister in nature, such as a computer virus or Trojan horse. Malicious code can threaten the integrity, confidentiality and availability of what was once thought-of as "trusted" software, which is the heart of modern military information management/transport systems. In a commercial-off-the-shelf/Common Operating Environment-based environment, code can be clandestinely embedded within a system's software suite at most any time: from development through segment/component integration, distribution and operation. The U.S. Army Communications-Electronics Command Software Engineering Center (SEC), in support of the U.S. Army Command and Control Protect Program, is actively involved with analyzing software to detect malicious code. This paper discusses SEC's activities – from the current, primarily manual analysis at the source code level, to plans for semi-automated tools and a discussion on existing research into fully-automated analysis of both source and executable code – for improving existing methods of malicious code analyses and detection.**

INTRODUCTION

Reference [1] lists Trojan horses, viruses, worms, insider attacks, hackers and phreakers as being the basic threats to military computer systems. Another bulletin [2] states that "although insiders cause more damage than hackers, the hacker problem remains serious and widespread."

The old concept of a computer system's vulnerability took the form of malicious code, such as viruses and worms and human threats both internal and external. These target the weaknesses of a system and are applicable yet today.

However, old concepts fail to come to terms with the new type of threat: a commercial-off-the-shelf (COTS) or free-ware/shareware product that has been produced with embedded malicious code or through coding practices that allow exploitation.

The new threat has arisen because of the expanded use of reusable code segments and the need for more economical ways to meet the user requirements; e.g., the use of a Common Operating Environment (COE). There has been a wider acceptance of generic COTS and freeware/shareware into secure environments. As a result, the vulnerability of "trusted" systems to embedded malicious code attacks has increased proportionally.

"Easter eggs," seemingly harmless and playful software embedded within other programs, are found in the most common business software within the Department of Defense, including mission critical defense systems. If it is possible to "sneak" this self-contained software into other software, why should it not be possible to add truly malicious components to other legitimate software? COTS software chosen for incorporation into defense systems has been programmed in nations known to have the capability and possible intent to initiate an information warfare attack on the United States. Additionally, software developers have been known to insert backdoors in software to aid in testing, and disgruntled employees have been known to set software time bombs after being fired. Software performs a trusted function as the heart of mission critical defense systems. There is therefore clear and present danger from malicious code.

Recently a L0pht Security Advisory [3], made this very apparent where a password appraiser package was sending the entire Microsoft® Windows NT® user password list across the internet while supposedly checking the strength of the users' passwords.

Another incident was reported in a recent Computer Emergency Response Team (CERT®) Advisory [4], where "…some copies of the source code for the TCP Wrappers tool (tcpd) were modified by an intruder and contained a Trojan horse … [which] provides root access to intruders initiating connections … [and] upon compilation, … sends email to an external address … [that] includes information identifying the site and the account that compiled the program."

These security issues are currently being addressed by the U.S. Army Communications-Electronics Command's Software Engineering Center (SEC) through a practice of malicious code analysis in support of the U.S. Army Command and Control (C2) Protect Program. The C2 Protect Program was established to improve the security posture of defense systems. While some stylistic, understandability or readability issues may be of concern, the focus of the analysis has been to identify software coding issues related to potential or concrete security impacts. Examples of these are provided in the following paragraphs.

A. Worms

Worms are programs designed to replicate themselves and cause execution of the newly copied version. A network worm copies itself to another system by using common network functions and then executes the copy on the new system. Worms replicate by exploiting flaws in the operating system or inadequate system management. They may cause a denial of service or gain unauthorized access by drastically reducing system resources, compromising confidential data, or causing unintended system operation.

B. Viruses

Viruses are programs that infect other programs by including a copy of the virus in the program. The virus contains unauthorized malicious instructions and may attempt to escape detection through polymorphism and other techniques. Viruses are replicated when the infected program is copied to another system via floppy disk, compact disk, electronic-mail attachments, or downloaded from the Internet.

Depending on the virus it may be launched when the file it is attached to is executed or when the system is booted from an infected boot sector.

C. Trojan Horses

Trojan horses are programs that contain hidden functions. They are often found in programs that otherwise provide a useful function. When the program is executed, the Trojan horse is launched, performing actions the user does not expect or want. Trojan horses do not replicate, they rely on users to install them and distribute them or intruders who have gained unauthorized access.

D. Hostile Mobile Code

Hostile mobile code – like Java Applets, common gateway interfaces (CGIs), or Active X – run and display inside an HTML web page. Although some security mechanisms are in place (e.g., constraining applet privileges within a "sandbox" or alerting the user based on authenticode certificates attached to Active X code), the mechanisms can be broken and the malicious code containing instructions that are damaging or unexpected can still be executed. They are activated when the associated web page is executed.

E. Backdoors

Backdoors provide access to accounts and files. Backdoors are established after the system's security has been breached and root access gained. Once a backdoor is established, it allows the malicious intruder to easily reenter the system further compromising data on the system, causing unintended operation, and allowing the "at-will" wholesale destruction of the system. This also allows a malicious user to use the system as a launching point to attack other systems or as a storage location in the future. It this way the malicious user can evade detection from the "end system" under attack.

F. Coding Errors

Coding errors that cause race conditions and buffer overflows can provide unauthorized access to the system. Race conditions occur when more than one process performs an operation and the result of the operation depends on unpredictable timing factors. A race condition can give a user the capability to write to a file when he normally would not be able. For example, a race condition occurs when one process is writing to a file while another process is trying to read from that same file. Buffer overflows occur when incoming data exceeds the storage space allocated, causing the return stack pointer to be overwritten. A buffer overflow allows the user to change the return address of a function and thus change the flow of program execution.

G. Standard Coding Practices

Standard coding practices may expose the system to vulnerability, whether the vulnerability is intrinsic to the coding method or known operating system weaknesses. Standard coding practices such as the use of *strcpy()* in the C language allows the user to use this function to copy a string from one array to another. The *strcpy()* function is vulnerable to buffer overflows. However, the C function *strncpy()* also copies a string from one array to another, yet it limits the total length of the string that is copied and is not vulnerable to buffer overflows when used properly.

There are many ways malicious code can be introduced into a system, either intentional or purely innocent. During software development, Trojan horses as well as race conditions and buffer overflows can be introduced to the software package. During installation of the system, viruses and worms can be introduced. Even when building the system, malicious code can be introduced via the compiler software or other tools. The developer may have produced a clean secure software package, but when the software package is installed or built using a malicious environment, the result could be damaging. Additionally, malicious code can be introduced during the distribution of the software. The distribution environment does not usually fall within the bounds of malicious code analyses, but it is important to note that the security of the distribution channel must be addressed in order to obtain the full benefit of malicious code analyses.

The remainder of this paper discusses three basic methodologies that SEC is currently using or researching for malicious code analyses: (1) manual source code analyses/review, (2) semi-automated analyses, and (3) fully automated analyses. Table I shows the cost, efficiency and advantages/disadvantages for each of the three basic methods.

The system to be analyzed must have software CM procedures and processes put into place. The baseline system

TABLE I
MALICIOUS CODE ANALYSES METHODS AND ATTRIBUTES

Method	Cost	Efficiency	Advantages	Disadvantages
Fully Manual	High	Low	• Staff maintains maximum knowledge of product internals. • Allows benefit of professional insight in code review.	• Code reviewers may suffer from "code reading fatigue," reducing their accuracy. • Team members may have differing levels of sophistication.
Semi-Automated	Moderate	Moderate	• Introduces efficiencies and traceability into the fully manual model. • May be supported or interfaced with fully automated support tools to identify specific threats present in source code, as well as known safe pieces of code. • No state-of-the-art breakthroughs are necessary.	• Still relies on manual efforts, subject to human error.
Fully-Automated	Low	High	• Based on known, documented principles. • May be able to discover new attacks or mechanisms based solely on models and templates of known hostile characteristics. • Repeatable.	• Will only provide results based on specific parameters. • Human insights may be lost.

must be identified, tracked, and controlled. The malicious code analysis process has no value without stringent CM practices being exercised during the development, building, distribution, and installation of the software product. The malicious code analysts themselves must keep the software being reviewed under constant CM control.

In all three methodologies, an initial analysis must be performed confirming that a complete system has been delivered. Once that effort has been accomplished, the malicious code analysis can begin.

MANUAL SOURCE CODE ANALYSES/REVIEW

The first step taken in the manual malicious code analysis is to run available static analysis tools, which include sizing, metrics, and textual-search tools. The sizing tool is executed to organize the code modules/files and provide a line of code count. The metrics tool is executed to provide a measure of the code complexity. The textual-search tool is executed to find commands, functions or phrases, (e.g., *chmod, strcat,* and */etc/passwd*) that alert the analyst to possible vulnerabilities in the source code. In addition, ad-hoc query scripts, debuggers, and trace statements can be used to locate and study particular operations and program flow. Also, the available system documentation is reviewed. Both tools and the documentation are used to aid in the task analysis breakdown and to gain in-house familiarity into the nature of the system.

After familiarity is gained with the overall system, each source file is analyzed line by line. Although not all inclusive, as in [5], the following is a sampling of the types of potential deficiencies looked for in the written code.

A. Password Protection

- Required passwords that are not properly safeguarded
- Code that sends passwords in the clear

B. Networking

1) Code that provides excessive access to files across the network
- Code that opens ports that do not need to be open
- Items that may connect to systems or software subsystems in an unsafe manner

C. File Permissions

2) Code that changes file permissions unnecessarily
3) Programs that take ownership of files that they should not
4) Programs that access publicly writeable files/buffer/directories with potential for malicious exploitation

D. Minimum Privilege

5) Code that does not prevent abuse of required access privileges
6) Code that is granted more than the minimum privileges necessary to perform its function
7) Programs that provide shell access; these should be considered suspect as they may be used to obtain excessive privileges

E. Self Replicating/Modifying

8) Code that self-replicates across systems
9) Code that is self-modifying

F. Bounds and Buffer Checks

10) Code that does not have proper bounds and parameter checks for all input data
11) Arguments that are not current and valid for system calls
12) Code that uses unbounded string copies/arguments; such code may be vulnerable to buffer overflows

G. Race conditions

13) Conditions where one process is writing to a file while another process is reading from the same location
- Code that changes parameters of critical system areas prior to their execution by a concurrent process
- Code that improperly handles user generated asynchronous interrupts
- Code that may be subverted by user/program generated symbolic links

H. Other Checks

14) Excessive use of resources
- Code that is never executed; such code may execute under unknown circumstances/conditions, and consume system resources
- Implicit trust relationships that could induce vulnerabilities
- Code that does not meet functional security claims (if the system purports to perform passwords/logs/security, does the code actually perform those functions)
- Code that performs a malicious activity
- Code that uses relative pathnames inside the program with a potential for accessing unintended files including dynamically linked libraries (DLLs)

SEMI-AUTOMATED ANALYSES

Performing a fully manual analysis of all source code in a large software system can offer significant if not total assurances that malicious or otherwise malevolent code was not injected into the software. However, the total line-by-line code review of systems may be prohibitive in both time and cost; worse yet, the effects of "code fatigue" on individuals charged with reading every line of code in large systems can degrade the assurances offered by a manual code review with the effect that a false sense of security may be realized. Included in this fatigue is susceptibility to psychological tricks that may be presented in the software comments.

Semi-automated approaches could make use of programs designed specifically to assist the code review analyst in the task of reviewing these systems, especially in identifying and locating potentially suspect program code. A program may be constructed with which to semi-automate the overall process; such a program could make use of the following four components: a target word list and its respective database facility, a database, a graphical user interface, and a report generation engine.

A target word list would be maintained; each line of code containing any of the "target" words would be flagged by the program as requiring manual inspection due to their explicit relevance to systems security. Examples of such "target" words (for Unix systems) would include keywords such as: */etc/password*, *chmod*, *chown*, *su*, and *chgrp*.

A program is needed to then build a database by reading the list of target words, as well as all program source code for the system or program under study. All program source files which contain one or more of the target words will be identified as such and would have appropriate database entries added automatically indicating the grounds for the suspect code to be treated with due suspicion, i.e. the specific target words which were identified. These entries would contain relevant information, including: file name, file type (e.g., C program file, C language header file, shell programming script), the line number and any associated target word(s) contained therein, a status flag describing the present state of the file (not inspected, hazardous, safe, or unknown), a comment field for analyst-generated comments and concerns, anomaly information, lines of code and complexity metrics. Date stamps will be incorporated to all record creations and modifications, so that their histories may be effectively logged and traced. A hybrid methodology between semi-automated and fully-automated analysis could incorporate a library of small independent programs, each capable of identifying a specific type of anomalous code, and injecting an appropriate record into the database.

A graphical user interface (GUI) is necessary for the searching, selecting and displaying of source code files, providing analyst annotations or comments relevant to the source code files, indicating or changing the recorded status of any files under study, and organizing the overall work into tasks for the group to conduct the code study.

The system will provide facilities through the graphical user interface to display any or all files of interest, most importantly and vitally, those not yet deemed safe. For each file, which is not "safe," its database field description will be provided. If the analyst clicks on any of these entries, the file corresponding to that particular database entry will be displayed, with the analyst automatically brought to the particular line of source in question; the analyst shall at this point be able to freely maneuver within the given file.

If the analyst wishes to declare the code "safe," the analyst will have the opportunity to do so via the user interface. Further, the analyst at this point may elect to mark the code as hazardous, suspect, or unknown, providing comments, or, may elect to take no immediate action. Any "action" shall be reflected by the system updating the database.

An efficient option may also allow for the analyst to select a fixed number of lines appearing both before and after the code in question, and for the system to automatically accept as safe all identical clusters of code. Such an option could hasten the reviewing of commonly repeated code segments, such as those found in file headers, or chains of similar and repetitive expressions.

Note that such user interaction may be readily provided either through a web-browser type of interface, which could make extensive use of hyper-links, or through a specific application developed to meet this particular need. The analyst will be able to maneuver freely through any code, or to follow the links until all potentially anomalous lines of code which were identified are determined by the analyst to be hazardous, safe, or unknown.

A report generation program, made available to the analyst through the graphical user interface would query the database. A report would include statistics for the entire program, including: total lines of program code and any other metrics of interest, the number and nature of all "target words" detected, and in what files, the number of detected potential anomalies which were determined to be "safe," or any other information of interest which may be obtained from the database records. Most importantly, the report generation program would consolidate descriptions of all hazards identified during the code review, facilitating a comprehensive analysis of their significance and of any necessary or corrective action to be taken.

FULLY-AUTOMATED ANALYSES

Although it can never be a magic bullet, a fully-automated tool that can scan executables as well as source code will greatly improve existing methods of malicious code analysis. Most existing tools are platform-specific and are limited by the fact that they use ad-hoc methods to detect known coding vulnerabilities only. A fully-automated tool should be capable of "testing" all software running on a platform, as anti-virus software does, looking for distinguishing characteristics that might be malicious code. It would be integrated within a user-friendly GUI control panel and be modular, allowing additional functionality as new threats are identified.

When the tool identifies potential "bad code," it would alert the analyst to make the final decision as to whether the code is really malicious – what may appear to be malicious may in fact be the requirement of the software; e.g., deleting all files in a "temp" directory set up by the program. This feedback can be used to aid in future automated analyses.

Very often, rogue or flawed code is not detected with simple test coverage methods. On the other hand, exhaustively testing software is not practical either. More efficient would be for the testing to just focus on global properties of malicious software. While utilities such as intrusion detection systems search for signature characteristics of an attack, this automated analysis tool must also consider characteristics of software vulnerabilities.

An existence of an exploitable vulnerability is a property. Properties can be behavioral specifications or sets of generic program flaws. The premise behind property-based testing is that testing programs can be made more efficient and automated if the analysis is concentrated just on the part of the software that influences the property.

Prototype analyzers have been successfully developed based on a property-testing approach to identify those properties commonly displayed by malicious code [6]. Operating both in a static environment and at run-time to take the dynamic environment into account, a mature tool could conceivably inspect the code and locate such operations as system calls, graph the data flows and processes, study the arguments (taking into account aliases), and highlight code that is suspicious. At the same time, a database of malicious code behaviors can be assembled and maintained to provide an even more comprehensive classification of code behaviors.

While acknowledging that it is impossible to develop a perfect means of detection, it is possible to target and identify a large percentage of malicious code behaviors. It is expected that this approach will lead to the development of a set of tools to also identify appropriate countermeasures for

specific properties that can be used to neutralize not one, but potentially many coding vulnerabilities. In addition, it is expected that this type of classification and code detection will facilitate forecasting characteristics of currently unknown vulnerabilities.

A. Components

The tool must be able to understand the software operations. Therefore, the source or executable should be translated into a readable, intermediate format (the executable can first be disassembled before the translation). Next, the tool could analyze the code variables to determine how variables are allocated in the program, and perform a data-flow analysis to determine relationships between variables.

Usually, only a small portion of a program impacts a given property. It is envisioned that the tool could "slice" a program with respect to a property to reduce its size during testing. Program slicing is an abstraction mechanism in which code that might influence the value of a given variable at a location is extracted from the full program [7]. Slicing isolates portions of a program related to a particular property, e.g., filename generation, and reduces it into a manageable size allowing confirmation of suspicious code.

Most changes to the security state of a program occur through system calls, so the slicing criteria should be closely related to these system calls. Predefined suspicious events, such as attempting to open files, or change or inspect file permissions, can be sliced. In other cases, the specifications are on program variables, which then become the slicing criteria. Code statements that are not reachable from main entry point can also be identified.

The tool could include a tracking database to record and generate statistics and reports related to all possibly anomalous and inherently vulnerable code as identified by the property-based tools. The database and its associated environment will use data generated by the tool to provide a series of hypertext links (or similar mechanism) to take the user to the point in the code where the perceived coding vulnerability exists. The human analyst would then have the opportunity to investigate the perceived vulnerability to determine and record whether it is indeed malicious, or instead innocuous, for later tracking and accountability purposes.

B. Properties

The tool would utilize a database populated with known coding vulnerabilities' properties. This information may be a set of signatures, a set of environmental conditions necessary for an attacker to exploit the vulnerability, a set of coding characteristics to aid in the scanning of code for potential malicious intent, or other data. The classification scheme should be well defined and unambiguous. Determining whether a vulnerability falls into a class requires either an "all yes" or "all no" answer. Similar vulnerabilities should be classified similarly, although, it is *not* required that they be distinct from other vulnerabilities. Classification should be based on the code, environment, or other technical details. This means that the social causes of the vulnerability are not valid classes. While valid for some classification systems, this information can be very difficult to establish and will not help in uncovering new vulnerabilities. Vulnerabilities may fall into multiple classes. Because a vulnerability can rarely be characterized in exactly one way, a realistic classification scheme must take the multiple vulnerability-causing properties into account.

Each vulnerability has a unique, sound property set of minimal size. Call this set the basic property set of the vulnerability. Determining the similarity of vulnerabilities now becomes an exercise in set intersection. Using the definitions and characteristics of the defined classifications and properties suggests a procedure to locate vulnerabilities: look for properties. When detected, the condition described by the property must be negated in some manner. All vulnerabilities with the same property are therefore non-exploitable.

CONCLUSIONS

Bringing code inspection into contemporary truly mission-critical military platforms with security in mind as a paramount concern does demand that certain costs be borne, but experience, inspections, and research suggest that economies and efficiencies of scale and operations can be achieved. Between the conventional, semi-automated, and fully automated approaches being currently developed, the facilities are being readied to ensure the trustworthiness of source code from a security perspective. Historically, and especially, in government domains, code inspections have been considered a viable approach in ensuring that software has possessed certain desirable characteristics, was devoid of certain undesirable characteristics, and was generally of high quality [8]. Malicious code analyses processes introduce an additional opportunity to present CM version control to COTS and after-market tools and software components which may be critical to secure military computing. After all, in the absence of CM version control and code review methods, project managers and their developers do not actually know precisely what has been installed on their platforms, or the threats unleashed on their system – do they?

REFERENCES

[1] L.E. Bassham and W.T. Polk, "Threat Assessment of Malicious Code and Human Threats (NISTIR 4939)," National Institute of Standards and Technology Computer Security Division, October 1992.

[2] "The Computer Systems Laboratory Bulletin," National Institute of Standards and Technology, March 1994.

[3] L0pht Security Advisory, L0pht Heavy Industries, 21 January 1999.

[4] "Trojan horse version of TCP Wrappers," *CERT Advisory CA-99-01-Trojan-TCP-Wrappers*, CERT Coordination Center, 21 January 1999.

[5] C.E. Landwehr, A.R. Bull, J.P. Mc-Dermott, and W.S. Choi, "A taxonomy of computer program security flaws, with examples," *Technical Report NRL/ FR/5542-93-9591*, Naval Research Laboratory, November 1993.

[6] G. Fink and M. Bishop, "Property-Based Testing: A New Approach to Testing for Assurance," *ACM SIGSOFT Software Engineering Notes*, Vol. 22, No. 4, July 1997.

[7] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, July 1984.

[8] D.P. Freedman and G.M. Weinberg, "Handbook of walkthroughs, inspections, and technical reviews:

evaluating programs, projects, and products," Dorset Publishing Co., NY, NY, 1990.