Malicious Code*

W. Eric Wong and Vidroha Debroy
Department of Computer Science
University of Texas at Dallas
Email: {ewong,vxd024000}@utdallas.edu

Malicious code is as big of a problem today as it ever was, if not larger still. Software is becoming increasingly more complex, and many software systems themselves do not operate in isolation but rather are connected and in fact sometimes dependent on other systems. An attack on a software system therefore is a potential attack on any other system that it may interact with, which further magnifies the damage. Many approaches have been proposed to deal with malicious code and its adverse effects, and these approaches have all met with varying degrees of success. A large part of the problem is that prevention and resolution strategies are always a step behind the creation and deployment of malicious code. Each manifestation of malicious code usually requires its own fix, and therefore there is no miracle cure that can help detect and prevent, if not nullify, all malicious code. To be able to analyze any piece of software and deem it free of all malicious code is as difficult as any other un-decidable problem. Now that we are aware of what we cannot do, let us try to briefly describe malicious code in general so that we can be in a better position to address what it is that we can do.

"Malicious code is any code added, changed, or removed from a software system to intentionally cause harm or subvert the system's intended function [1]." It is however important to acknowledge that when fighting malicious code we care not of its intention but rather on its effect on the software. Code inserted with the most benign of intentions can have the most malicious of effects. Take, for example, programmer negligence that manifests itself in the form of a buffer overflow. Previously, some of the more important sources of malicious code were third party renovation and remediation, off the shelve commercial/non-commercial systems that may spread pre-existing malicious code, and disgruntled employees/contractors or anyone else that might have access and the ability to insert malicious code into the system. However, with the popularity of the Internet and the growing inter-connectivity of computers, systems are vulnerable to attacks, with or without human intervention, from just about anywhere regardless of physical distance or connectivity. Due to such connectivity, an attack can be propagated to a large number of machines in a relatively small amount of time, which may in turn propagate the attack to other machines causing a chain reaction. This ripple effect makes it equally as difficult to re-trace the attack back to its source, and there is no reason to believe it shall be easier to do so in the near future.

Several categories and classifications exist to help narrow down the type of malicious code that one might be dealing with. The general categorization of malicious code has been along the lines of worms, viruses, Trojan horses, time bombs, backdoors, etc. [2]. The literature on malicious code has several definitions and examples of each category. Further classification might be done based on the mode of propagation, the nature of the attack, the portion of the system targeted, etc. For example, the presence of malicious code may cause different levels of damage to the normal operations of a system such as disruption of a non-critical subsystem, disruption of a critical subsystem, crashing of the entire system, granting unauthorized access, re-directing sensitive data, deleting sensitive data, etc. Risk-based models exist that try to decide on an appropriate

* This article is part of the IEEE Reliability Society 2008 Annual Technology Report which was published in *IEEE Transactions on Reliability*, Volume 58, Number 2, Pages 249–251, June 2009.

course of action based on risk evaluations for each category. Identification of low or no risk components allows concentration on essential, high risk components. Furthermore, a methodology that indicates application areas that exhibit high probabilities of potentially containing malicious code allows for selective analysis of such areas based on their proneness or vulnerability. A combination of these models might cause a tradeoff between areas with high proneness but low risk, and those of high risk but low proneness. However, if there are areas of high risk and high proneness, then that is where the maximum of our efforts to handle malicious code should go. Typically, these areas consist of only a small percentage of the entire system. Intense effort focused on such areas will pay the largest dividends.

Malicious code is quite hard to test for. Let us first assume that we are developing a piece of software and one of our disgruntled programmers inserts a piece of malicious code (such as a time bomb) into it. This code does not cause the software to deviate from its specification in any way for a pre-determined amount of time or until a counter variable reaches a certain value. No test case that checks to see if an output matches an expected output will be able to detect such malicious code as its effects are unobservable, at least for the time being. Given that we cannot detect the effect of the code, a test case would have to be able to detect the actual presence of the malicious code in order for us to even know that it exists in our software. Alternatively, we may be able to simulate the passing of system time, or increment each counter variable present to its maximum value in order to detect the time bomb. However, each of these requirements would require a massive overhead in order to design and execute a large number of test cases that might possibly reveal the inserted malicious code. It should therefore come as no surprise to us that so many vulnerabilities of software to attacks are exposed well after the software has been tested. That being said, it is also useful to point out the inherent similarities that exist between malicious code detection and fault localization and debugging. Each area that might come under attack can be assigned a numerical score in order to evaluate its riskiness. Therefore, we can impose an order in which to test various areas and have a rough idea of how much we want to test them based on their numerical score. This is similar to fault localization where statements are ranked in order of their suspiciousness (computed by some heuristic [3], [4]) such that statements with higher suspiciousness are examined first. This kind of analysis is also very similar to the analysis of safety and/or security critical systems such as nuclear power plants or flight controller software. In the case of such safety dependent systems, a software hazards analysis is often done in order to identify failure modes that could lead to an unsafe state [5]. Finding and eliminating malicious code at the source level, has the overwhelming advantage that no damage is done to the data or the system, with the exception of any damage incurred while detecting the malicious code.

Several approaches do exist that perform various kinds of analyses on source code or on the corresponding executable. Some rely on structural features of executables that are likely to indicate the presence of inserted malicious code. The underlying premise is that typical application programs are compiled into one binary, homogeneous from beginning to end with respect to certain structural features; any disruption of this homogeneity is a strong indication that the binary has been tampered with [6]. Others have employed data-mining techniques [7] to identify patterns to detect malicious binaries and machine learning techniques such as support vector machines [8] to detect computer viruses. A large chunk of the research work on malicious code detection focuses on static analysis of the code. Static analysis deals with examining program code to determine properties without actually executing any of the code in question. In [9] the approach to detect malicious code is based on program slicing. High-risk patterns are defined and then used as the basis of a forward and backward static program slice. In [10] static analysis techniques are applied to binary code. First, the binary code is translated into an intermediate form, then the intermediate form is abstracted through flow-based analysis as various relevant graphs, and finally these graphs are checked against security policies. However,

static analysis techniques have some inherent limitations in that they make certain assumptions that could be un-checkable statically such as behavior with respect to array bounds and pointer aliasing. Some such limitations are pointed out in [11] where it is demonstrated that static analysis techniques alone might no longer be sufficient to identify malware. However, relatively little work has been done on utilizing dynamic analysis either alone or in conjunction with static analysis in order to effectively detect malicious code.

Dynamic analysis aims to test and evaluate a program by actual execution of code in real time quite possibly with some input. Dynamic analysis allows us to artificially create situations where the software is more likely to fail and thereby assess how well our program does what it is supposed to do practically as opposed to theoretically. It is true that static analysis holds some advantage over dynamic analysis. First, static analysis does not require any overhead in terms of test case execution and test case processing. Second, any results provided by static analysis hold invariantly and are test case independent. The results of dynamic analysis can in some cases be very closely linked to the choice of test cases employed in one's suite. However, dynamic analysis using dynamic slicing can reveal program properties in terms of program behavior that cannot be revealed by static analysis alone. Some malicious patterns might exist that cannot be exposed by static analysis techniques but might be revealed by one simple execution. All things being equal, the solution is fairly obvious – static analysis techniques and dynamic analysis techniques must work together in tandem. The amount of each analysis and the depth of the analysis would of course depend on the resources available. In [12] an automatic malicious code analysis system is proposed that tries to integrate the advantages of static and dynamic analysis as well as that of network behavior analysis. Such techniques are likely to prove very useful in the detection of malicious code as they take into account more than what can be done by any singleton approach. The application of combined static and dynamic analysis techniques to software is expected to be a rich area of research for some time to come.

While the aforementioned deals with analysis of the code to identify malicious code, there exist other techniques to deal with it. A host can also protect itself from malicious code by re-writing the code, rendering it harmless, monitoring the code and stopping it before it does any harm, and auditing the code and taking appropriate policing action if the code does do some harm [1]. Techniques such as appropriate security policies, solid encryption, code-signing, etc., are also fairly popular. Of late, another consideration is that most malicious code in large scale software systems is placed by insiders with access to code. The field of biometrics analyzes characteristics and traits of personnel, and this information is used to evaluate how much access they should have to critical portions of a system. However, this approach is less along the lines of detecting malicious code and more along the lines of detecting malicious coders.

Finally, no discussion could be considered complete without at least talking about some of the major obstacles and concerns regarding malicious code today.

(1) To develop new, and extend current, models to detect malicious code that take into account both static and dynamic analysis techniques and to construct and formulate the model such that it is cost effective and highly accurate.
(2) To develop better approaches that allow for tolerance of malicious code. Steps need to be taken to avoid system failure in the presence of malicious code such that its malicious effect can be mitigated and its propagation arrested.
(3) To develop lightweight techniques that can dynamically update a system such that it is not affected by the same attack again.
(4) To develop suitable testing criteria and test case generation techniques for testing with the intent of revealing malicious code.

These are expected to be the key areas of research interest in the field of malicious code prevention, detection and handling for some time to come.

References

[1] G. McGraw and G. Morrisett, "Attacking malicious code: a report to the Infosec Research Council," *IEEE Software***,** Volume 17, Issue 5, pp. 33 – 41, September 2000.

[2] D. M. Kienzle and M. C. Elder, "Recent worms: a survey and trends," in *Proceedings of the 2003 ACM workshop on Rapid malcode*, pp. 1 - 10, Washington, DC, USA, October 2003.

[3] W. E. Wong, T. Wei, Y. Qi, and L. Zhao, "A Crosstab-based Statistical Method for Effective Fault Localization," in *Proceedings of The First International Conference on Software Testing, Verification and Validation*, pp. 42-51, Lillehammer, Norway, April 2008.

[4] W. E. Wong, Y. Shi, Y. Qi, and R. Golden, "Using an RBF Neural Network to Locate Program Bugs," in *Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering*, pp. 27-38, Seattle, Washington, November 2008.

[5] R. G. Huget, M. Viola and P. A. Froebel, "Ontario Hydro experience in the identification and mitigation of potential failures in safety critical software systems," *IEEE Transactions on Nuclear Science*, Volume 42, Issue 4, Part 1-2, pp. 987-992, August 1995.

[6] M. Weber, M. Schmid, M. Schatz and D. Geyer, "A toolkit for detecting and analyzing malicious software," in *Proceedings of the 18th Annual Computer Security Applications Conference*, pp. 423-431, Las Vegas, Nevada, December 2002.

[7] M. G. Schultz, E. Eskin, E. Zadok and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 38-49, Oakland, California, May 2001.

[8] B. Zhang, J. Yin, D. Zhang and S. Wang, "Using support vector machine to detect unknown computer viruses," *International Journal of Computational Intelligence Research*, Volume 2, Number 1, pp. 100-104, 2006.

[9] W. Raymond, L. Karl and O. Ronald, "MCF: A malicious code filter," *Computers and Security*, Volume 14, Issue 6, pp. 541-566, 1998.

[10] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie and N. Tawbi, "Static detection of malicious code in executable programs," in *Proceedings of Symposium on Requirements Engineering for Information Security*, Indianapolis, Indiana, March, 2001.

[11] A. Moser, C. Kruegel and E. Kirda, "Limits of static analysis for malware detection," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, pp. 421-430, Miami Beach, Florida, December 2007.

[12] J. Zhang, Y. Guan, X. Jiang, D. H. Duan and J. Wu, "AMCAS: An automatic malicious code analysis system," in *Proceedings of the 9th International Conference on Web-Age Information Management*, pp. 501-507, Zhangjiajie, China, July 2008.