Security: Where Testing Fails

Cynthia E. Irvine

Department of Computer Science

Naval Postgraduate School
Monterey, CA 93943

Computer security addresses the problem of enforcement of security policies in the presence of malicious users and software. Systems enforcing mandatory policies can create confinement domains that limit the damage incurred by malicious software executing in applications. To achieve assurance that the confinement domains cannot be breached, the underlying enforcement mechanism must be constructed to ensure that it is resistant to penetration by malicious software and is free of malicious artifacts. The limitations and contributions of testing in achieving these goals are discussed.

Why would a national software testing laboratory advertise on its web page that it provides testing for functionality, compatibility, performance, scalability, and fault tolerance, but not security?

The answer may lie in the fact that certain aspects of security policy can be described in completely non-subjective terms. For example, the policy may state that unauthorized individuals are not permitted to read classified material. Can testing ensure that policy will not be violated? This paper provides an overview of challenges that security poses to testing and describes the role of testing in the engineering of trustworthy systems.

System Security

In fundamental terms, information security is concerned with the control of access by individuals to information in accordance with an organizational policy. That policy will describe which users are authorized to read particular information and which users are authorized to modify that information. Organizational security policies have two dimensions: confidentiality and integrity. Rules, laws, and directives describe criteria for identifying information to be protected and individuals authorized to access that information, with clear guidance so that individuals can know when policy is being violated (Sterne 91).

Information is protected in order to maintain secrecy, as might be necessary for military data or for proprietary corporate information, e.g. the secret formula for a highly popular soft drink. To ensure the reliability of information, its integrity must be protected by permitting modification only by authorized individuals. The reliability (or integrity) of information must be maintained because critical decisions or actions may depend upon it. An example might be a digitized x-ray indicating the location of a brain tumor to be removed. Malicious modification of the image could prove fatal.

In the era before computers, one might choose to prohibit unauthorized access to information through a variety of procedures and practices. Thus secrets were stored in a safe and the safe was kept in a special room. Access to the room was controlled by a guard or keys, and access to the contents of the safe was controlled by issuing the combination only to those users having a *bona fide* need to access its contents.

Security in computers is an extension of attempts to control the access by people to information. The use of computers enriches the notion of information to include not only the original information that existed outside of the computer, but many forms of information that are particular to the use of networked systems. Some of these are: programs, authentication information, cryptographic keys, system databases, application databases, as well as information in formats ranging from binary data through e-mail and web pages. Our notion of people is also extended. In the cyber-world, processes act on behalf of people and these processes can be assigned attributes so that the are identified with the individuals who initiate them.

Security Threats

Threats to computer security cover a broad range of problems. Among the most common are unintentional errors on the part well-meaning users. Mistakes must be addressed through a combination of user training and well-designed user interfaces.

A more insidious threat is that of malicious software. In this case the user does not make mistakes and the interface may be excellent, however the software is misbehaving despite the user's good intentions. Recall that in computers processes act on behalf of individuals. These processes execute programs and have no judgement, they merely proceed through a set of instructions and branch depending upon current state and inputs. If the software has been corrupted it will not behave as intended, but the users are not in the computer to prevent what may be a grievous error.

Malicious software may be hidden within a desirable utility or application, and it may, in fact, perform desirable functions; however, behind the scenes the software is exploiting the privileges of its user to steal secrets or corrupt reliable information. Malicious code that piggy-backs its way into systems is known as a *Trojan Horse*. The Trojan Horse taxonomy is quite varied and the threat of Trojan Horse contamination has been exacerbated by rapid increases in connectivity that have not been complemented adequate security. The use of downloadable code and scripts has rendered both e-mail and web browsing vehicles for malicious code insertion that can cause grave damage to organizations through unauthorized access to information.

Malicious code downloaded after the system has been built is a serious problem, but perhaps an even more insidious form of malicious code is code implanted during the development process intended to subvert the system itself. Here the mechanism that is intended to support system security policy enforcement is altered to meet an adversary's objectives.

Subversion can add an artifice to a system that might be triggered, for example by a string containing the phrase "attack at dawn." At this point the system might turn its control over to a hostile entity that provides instructions to make the system misbehave or turn on its owners. In his Turing Prize lecture, Ken Thompson described a trapdoor inserted into the Unix operating

system that permitted him to take over the system (Thompson 84). Through clever modifications to the C language compiler, he was able to eliminate all traces of the artifice from the source code of both the Unix operating system and the compiler. It was essentially impossible for others to find the trapdoor even after being told about it. Thompson's artifice continued a long tradition of system subversion that originated with Tiger Teams of the 1970s (Anderson 72).

Applications can also be subverted, as are many popular commercial software products. Such subversion often takes the form of code added by the development team so that its members can be given credit for their work. (A particularly elaborate subversion consists of a flight simulator embedded in a popular spreadsheet application in which the user can hover over a slab upon which the names of the developers are scrolled.) Entire websites are devoted to cataloging these artifices, often referred to as *Easter Eggs*. The existence of such rampant subversion of products upon which government and industry depend should give the reader pause. What about the subversion that is not so benign: the artifices implanted by hostile entities and intended to do harm?

Myers describes many points during a system lifecycle that afford opportunities for system subversion (Meyers 80).

Policy and Requirements. If the policy to be enforced is flawed, then even though the system may be implemented perfectly, it will never be secure. System requirements may be described so that protection mechanisms are incomplete, leaving ample opportunity for subsequent penetration and takeover.

Design and Implementation. High level designers might recommend security solutions that are inadequate. For example, the designer may send the team off to build an elaborate intrusion detection mechanism knowing full well that it will be easy to slip a trapdoor into the operating system which appears to be tangential to the security engineering team's focus.

Testing. During testing, flaw remediation could add trapdoors while repairing other problems. Alarming rumors of trapdoors inserted during Y2K remediation provide an example of the possibilities in this area. In addition, testers could be adversarial and avoid certain tests.

Distribution. An adversary could intercept the system during the delivery process and modify the system or substitute a flawed version. Even systems that had been developed in a highly controlled environment could be subverted. How can customers who acquire software through third-party distributors be certain that they are receiving the product as developed by the vendor?

Installation. During installation, the clever adversary can introduce subversive elements by providing bad advice and bad configurations. In many cases systems are so complex that consultants are brought in to set them up. System owners must have confidence that the system is installed and managed in a trustworthy manner.

Updates. The use of system updates to repair flaws or increase functionality is increasingly common. Since many commercial products are delivered with numerous bugs and are, in effect, tested by the customers, it would be simple to provide updates and patches to insert a trapdoor.

Unless carefully controlled, the update process may provide an adversary with ample opportunity to insert an artifice into a formerly trustworthy system.

Malicious software can be characterized as a class of unintended function. From the perspective of software engineering, unintended function can result from errors as well as intent. Can testing help to eliminate unintended function?

Searching for Malice

System developers have functional objectives that must be met. For example, the system must support banking transactions or it must track targets with specified location accuracy and timeliness constraints. These requirements specify what the system *will do*. Those concerned with computer security seek assurances of what the system *will not do* as opposed to what it will do. Knowing that the code carries a proof with it that guarantees how it will execute is insufficient for security purposes.(Of course, security is not sought in a vacuum and the system must meet functional as well as security requirements, otherwise we would have the proverbial "secure brick.")We begin with an example to illustrate the testing problem.

Suppose that the identification and authentication mechanism for a system has been subverted. When the user name "Evl1fMRY" followed by the password "23Bd-4u!" are entered, the user is given privileges that will permit access to anything in the system. Of course the name and password are encoded in the software, not the password database. In addition, the clever adversary has not left the name and password as simple strings but has instead hidden them through encryption or some other technique. In addition, the artifice is not hidden in the identification and authentication module but is driven by a modified version of the sound card driver which peeks at the keyboard input character stream. Can this artifice be discovered through testing?

Consider blackbox testing first. Here the tester only has access to the interface and treats the mechanism like a blackbox: parameters go in and results come out. The test is to determine if there exists a character sequence that would permit control of the system with unconstrained privileges. We have a name consisting of 8 ASCII characters, so there are 10^{19} possible names that could be input (assuming both printable and unprintable characters and that characters can be repeated). Similarly, there are 10^{19} possible eight character passwords. This means that to ensure that a particular combination of eight character name and eight character password will not yield total control of the system, 10^{38} tests would be required. Suppose that tests could be run at a rate of 10 million per second. A total of 10^{15} tests could be performed per year on a fast multiprocessing machine. Even with a billion such machines the testing would take 10^{14} years to complete--10,000 times longer than the current age of the universe.

The example above demonstrates a variant of Dijkstra's famous statement (Dijkstra 76): "Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence."

Could whitebox testing help? Here the testing team has access to the system. If the adversary is trying to hide an artifice within a large, complex system such as an operating system or virtual machine, it will be possible. One tiger team operating in the 1960s even told the

operating system developers that they had inserted an artifice into their system and the developers could not locate it.

We can reuse Dijkstra's words to state that program testing can be used to show the presence of malicious artifices, but never to show their absence. From a mathematical perspective testing for trustworthiness of a system is an intractable task (Hamlet 89).

What development process can address the problem of subversion and what is the role of testing in ensuring that the security policy enforcement mechanisms themselves are not subverted? In the next section, we will outline engineering approaches that support the development of systems for which one can have high confidence of the absence of subversion.

Security Engineering

With the vast amount of software being produced today, one might ask if rigorous security development practices must be applied universally. Fortunately, the answer is no. Only those portions of the system responsible for enforcement of the most critical protection polices need to be engineered with such care. These policies are termed mandatory because they are both global and persistent: they apply everywhere at all times. Government policies regarding the protection of classified information provide a well-known example of a mandatory policy. Its mandatory nature can be understood in terms of the consequences of its violation and the penalties imposed upon violators.

The goal of the security engineer is to create confinement domains that will prevent malicious software from reading or modifying information contrary to organizational policy. This will prevent unauthorized violations of confidentiality and will also prevent unreliable software from accessing critical data and software.

An early step in the security engineering process is to develop a mathematical model of the security policy that describes how active system entities can access passive information repositories while adhering to the system security policy. This formal security policy model demonstrates that the policy is not flawed and that the set of primitive operations described in the model do not violate security policy. The proofs involving low-level active and passive entities and the primitive operations of the formal model provide a cornerstone for subsequent system verification. During system development, a chain of evidence will be developed to demonstrate that the implementation is faithful to the model.

To provide guidance to system developers, an ideal mechanism for security policy enforcement, called the *Reference Monitor Concept*, has been described (Anderson 72). The Reference Monitor Concept describes a security policy enforcement mechanism having the following properties:

- The mechanism is tamperproof.
- The mechanism always invoked.

- The mechanism small enough to be subjected to analysis of its correctness and completeness with respect to its policy enforcement mechanisms as well as its penetration resistance and non-bypassability.

With this ideal in mind, developers will wish to focus resources on mechanisms intended to enforce the most critical, viz. mandatory, confidentiality and integrity policies. Careful engineering is needed to minimize the security enforcement mechanism and provide confidence that it is and will remain subversion-free.

An entire lifecycle process must be defined to ensure that the product will be trustworthy and free of *unintended functionality*. Elements of this development process are described below.

Configuration Management. A configuration team must be established and charged with maintaining the integrity not only of the evolving product, but also of the development environment--all of the tools used by the engineering team. The environment itself should be organized to prevent unauthorized development activities by insuring the accountability of development team members. Tools should be selected for their trustworthiness and maintained under configuration management.

Reviewed Design. By creating system documentation that is subjected to group review, members of the engineering team can ensure that design meets the requirements and does not contain malicious elements. A stepwise refinement process starting with high level system requirements documents and proceeding through design and coding reviews will contribute to the assurance of the process. Standards for the engineering development process as well as for engineering documents and code provide consistency and enhance inspectability of the implementation by team members. Code can be examined to insure that it meets its specifications and does not introduce artifacts.

Formal Methods. For the most critical system, formal methods can be used to verify that the system maps to the formal security policy model. Formal top level specifications can be proved to map to the model and thus can implicitly establish that those specifications are proved with respect to system security properties. Code correspondence can demonstrate that all code is traceable to the high-level specifications. As tools improve the possibility for formal verification down to the code level for these non-trivial systems may improve.

Trusted Distribution. The developer must ensure that the product and all subsequent updates are delivered to customers in a manner that prevents system corruption. Techniques to support trusted distribution range from the use of couriers to the use of cryptographic techniques involving digital signatures.

Independent Assessment. Even though the development team may claim that the product provides security policy enforcement with some level of assurance, customers must be wary of unsubstantiated marketing claims. Highly critical systems should not, and sometimes cannot, be examined in court after a disaster incurring loss of national sovereignty or catastrophic financial loss. Independent and unbiased evaluation of vendor claims is needed.

Trusted System Management. Users have a responsibility to ensure the continued trustworthiness of their systems. System administrators and system security officers must be properly trained to understand how site security policies can be enforced by their system, and how to maintain the system in a proper configuration. All users should be properly trained. System owners should understand the limitations of the system and where other security controls will be required.

Implicit in these aspects of system development and use is the notion of accountability. No unauthorized personnel should be permitted to modify the system and the contributions of authorized personnel should be auditable and traceable throughout the system lifecycle.

This development process is not perfect. Flaws may remain, but these flaws are likely to be extremely difficult to exploit. Thus the workfactor for violating a system will be raised to a level such that amateurs will seek amusement elsewhere and organized adversaries may find other methods of attack, such as the use of traditional espionage, substantially less costly and more productive.

The Role of Testing in Security Engineering

Testing is an important aspect of the secure system development process and consists of two parts: functional testing and penetration testing.

Functional Testing

During system implementation, functional testing will help to ensure that each module's specification is met. Such tests will provide evidence to independent evaluators that the system is free from obvious flaws. The development team will construct a test plan, defining test conditions based upon a security policy model for the system. To ensure adequate coverage, test data are carefully chosen. Ideally, one would want to test every branch with every possible input to each function. In a complex system, this is impossible, so the careful design of limited tests is required.

The functional testing team should plan and document a testing process that can be reviewed for flaws and omissions. Mechanical testing can aid the test team, but should produce results that can be usefully reviewed. To contribute to completeness, a test plan should be developed that describes how testing will map to the code and specifications of the target system. All test documentation including records of all tests should be available for review by observers independent of the test team.

Penetration Testing

Elaborate systems with rich policies have been designed; however, if the mechanism for this policy enforcement can be penetrated and subverted, then these policies will be ineffective. Thus secure system development also includes penetration testing. The Flaw Hypothesis Methodology (FHM) has been found to be an effective approach to penetration testing. Here independent evaluators test the system. The assurance of correct protection policy enforcement gained in penetration testing based upon the FHM is directly proportional to the expertise of the team

conducting those tests. Members must have expertise using the target system and experience in security testing. Weissman describes four stages of FHM testing (Weissman 95).

Flaw Generation is the first stage of the penetration exercise. Here the team creates a list of hypothesized flaws. The team brainstorms to create the list. Some of the factors, taken from (Weissman 95), that contribute to the list are: past experience found in the literature and of the team; ambiguous architecture and design; mechanisms that bypass of security controls; incomplete interface design resulting in implicit sharing which creates unintended information flows; deviations from the protection policy and model; deviations from initial conditions and assumptions; system anomalies and special precautions; operational practices, prohibitions and spoofs; development environment, practices and prohibitions; and implementation errors.

Flaw confirmation is conducted by organizing the flaws into a priority ordering based upon each flaw's probability of occurrence, gravity if confirmed, the level of work required to confirm its existence, and the area of the system design to which the flaw is related. Rankings are dynamic since as flaws are discovered others may become more or less important or likely. To confirm flaws, the team uses a combination of desk checking and live testing. The latter is costly and is most often used to confirm time-dependent synchronization or complex flaws.

Flaw generalization involves the reexamination of confirmed flaws to determine if they might point to additional flaws or even an entire class of flaws. Exploitable interactions between flaws are investigated.

Flaw remediation is recommended by the penetration team and might include anything from simple repairs to suggestions for system redesign.

Clearly, the team could work forever on penetration testing, so before starting a penetration exercise, the team defines their goals. This might be to work until one flaw is found, work for three weeks, or some other well defined notion of success.

Summary

The creation of secure systems presents significant engineering challenges. Instead of ensuring that a system behaves as specified, security engineers must address the problem of assuring that the system will not behave in an unintended manner. The malicious adversary is posited to be both sophisticated and well supported. It is impossible to propose tests that will ensure that a complex system is free of subversion, instead testing contributes in two ways: through traditional functional testing and through carefully designed, though necessarily incomplete, penetration testing.

A set of rigorous system development techniques can contribute to the implementation of highly trustworthy components for the enforcement of critical security policies. Substantial security engineering is required to ensure that these components are placed in an architecture such that they will, in fact, provide the desired enforcement of essential protection policies. Without high assurance components, malicious software executing in applications can rather easily penetrate systems to insert artifices, exfiltrate sensitive information, and corrupt information used for critical decisions.

Dr. Cynthia E Irvine is Director of the Naval Postgraduate School Center for Information Systems Security Studies (NPS CISR) and Research and an Assistant Professor of Computer Science at NPS. She has worked in the area of computer and information security for over thirteen years both in industry and in academe. Her primary research is on issues of information assurance and the recent focus of her research has been in the area of security for networked systems. She is a member of the ACM and a Senior Member of the IEEE. She received the Outstanding Research Achievement Awards from the Naval Postgraduate School.

References

Anderson, James P. (1972). Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA. (Also available as Vol. I, DITCAD-758206. Vol II, DITCAD-772806).

Dijstra, Edsger W. (1976). *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice Hall.

Hamlet, Richard (1989). Testing for Trustworthiness. In J. Jackey and D. Schuler (Eds*.), Directions and Implications of Advanced Computing*, pp. 97-104. Norwood, NJ: Ablex Publishing Co.

Myers, Phillip (1980). *Subversion: The Neglected Aspect of Computer Security*. MS thesis, Naval Postgraduate School, Monterey, CA.

Sterne, Daniel F. (1991). On the Buzzword "Security Policy". In *Proceedings 1991 IEEE Symposium on Research in Security and Privacy*, Oakland, pp. 219-230. Los Alamitos, CA: IEEE Computer Society Press.

Thompson, Kenneth (1984). Reflections on Trusting Trust. *Communications of the A.C.M*. 27(8), 761-763.

Weissman, Clark (1995). Penetration Testing. In M. Abrams, S. Jajodia, and H. Podell (Eds.), *Information Security: An Integrated Collection of Essays*, pp. 269-296. Los Alamitos, CA: IEEE Computer Society Press.