Addressing Software Security and Mitigations in the Life Cycle

David Gilliam & John Powell
California Institute of Technology, Jet Propulsion Laboratory
david.p.gilliam@jpl.nasa.gov,
john.d.powell@jpl.nasa.gov

Eric Haugh & Matt Bishop
University of California at Davis
haugh@cs.ucdavis.edu,
bishop@cs.ucdavis.edu

Abstract

Traditionally, security is viewed as an organizational and Information Technology (IT) systems function comprising of firewalls, intrusion detection systems (IDS), system security settings and patches to the operating system (OS) and applications running on it. Until recently, little thought has been given to the importance of security as a formal approach in the software life cycle.[1] The Jet Propulsion Laboratory has approached the problem through the development of an integrated formal Software Security Assessment Instrument (SSAI) with six foci for the software life cycle.

1. Introduction

The NASA Office of Safety and Mission Assurance (OSMA) has funded the Jet Propulsion Laboratory (JPL) with a Center Initiative, "Reducing Software Security Risk through an Integrated Approach" (RSSR), to address this need. The Initiative is a formal approach to addressing software security in the life cycle through the instantiation of a Software Security Assessment Instrument (SSAI) for the development and maintenance life cycles.

This SSAI, to date, has six elements: 1) a Vulnerability Matrix (VMatrix), 2) a Model-Based Verification (MBV) instrument with a Flexible Modeling Framework (FMF) that uses Model Checking and the SPIN model checker to check for properties in the requirements specifications that lead to vulnerabilities or unwanted exposures; 3) a Property-Based Tester (PBT) for JAVA and C Code for verification of security properties to verify the code that violations of these properties have not been re-introduced into it; 4) a Software Security Checklist (SSC) having two phases: Phase 1 addresses the development and maintenance life cycles, and Phase 2 addresses the external release of software; 5) a list of Security Assessment Tools (SAT's); and 6) Formal training for software and system engineers on software security and the use of these and other tools and instruments in the life cycle.

2. Vulnerability Matrix

The VMatrix is a vulnerability database whose purpose is to provide information about various vulnerabilities including exploits used to gain access to systems, how to protect against the exploits and the Common Vulnerabilities and Exposures (CVE) listing. The information is being transferred to the UC Davis Database of Vulnerabilities, Exploits, and Signatures (DOVES) where it will be maintained and updated as new exploits are discovered. This information is used to extract properties and requirements that express potential network vulnerabilities. These properties can then be utilized by the PBT and the FMF.

3. Security Assessment Tools (SATs)

The SATs are a collection of publicly available software security code checking tools available on the Internet that can be used to test for potential weaknesses in software code. The SATs are updated as additional tools become available.

The SATs are classified and grouped according to purpose and life cycle use. Included on the web site is a description of each tool and its intended application. Additionally, each tool's advantages and limitations are provided. Alternate tools are also included in this assessment including links showing where to obtain the tool.

Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop (SEW'03)
0-7695-2064-2/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

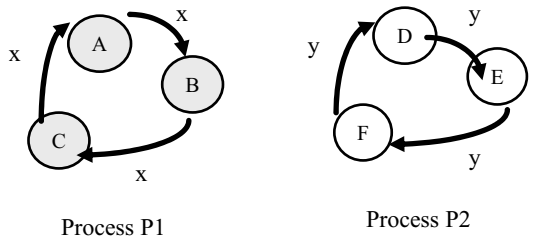

Figure 1: Concurrent Processes

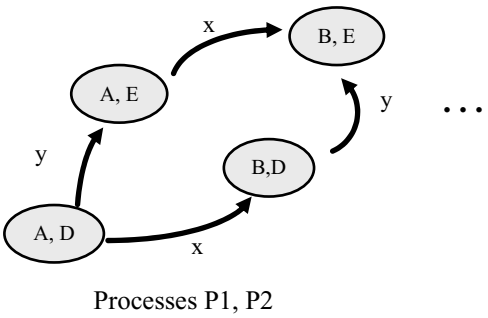

Figure 2: Interleaving Concurrent Processes

4. Model Checking (MC) and the Flexible Modeling Framework (FMF)

Software model checkers automatically explore all paths from a start state in a computational tree that is specified in an MC model. The computational tree may contain repeated copies of sub-trees. State of the art Model Checkers such as SPIN exploit this characteristic to improve automated verification efficiency. The objective is to verify system properties with respect to models over as many scenarios as feasible. Since the models are a representation of all functional capabilities under analysis, the number of feasible scenarios is much larger than the set that can be checked during testing. Model Checkers differ from traditional formal techniques by the following characteristics:

- Model checkers are operational as opposed to deductive
- Model checkers provide counter examples when properties are violated (error traces)
- Their goal is oriented toward finding errors as opposed to proving correctness since the model is correct

For example, consider a software system containing two concurrent processes with three states each. (See Figure 1) Process 1 (P1) is driven by an "X" event or input and process 2 (P2) is driven through its states by a "Y" event/ input. A model checker will automatically explore all possible paths through this system to determine if a given system property holds.

Testing of concurrent software systems quickly become infeasible as:

- The possible number of concurrent processes increases
- The functionality in one or more processes grows
- The interactivity between or complexity of one or more processes increases

This is largely due to the exponential growth in the operational state space of the software system in response to any one of the above stimuli.

This phenomenon is also, to a lesser degree, a limitation specific to model checking referred to as the state space explosion problem. [2] Similar to the growth of the operational space mentioned above, the state space that a model checker must search to verify properties grows at an exponential rate as the model of the software system necessarily becomes more detailed. Continuing with the example from figure 1, Figures 1 through 3 illustrate how the state space grows at a rate of m^n where m is the range of possible values a variable may assume and n is the number of variables in the model.

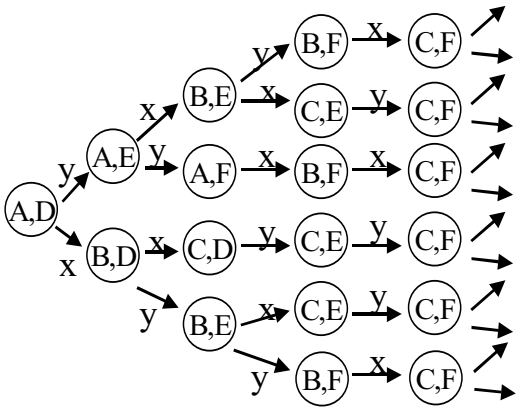

Figure 3: State Space

Despite the use of modeling techniques such as abstraction and homomorphic reduction, it is infeasible to verify many software systems in their entirety though model checking that are more than "moderately" large or complex.

An innovative verification approach, which employs MC as its core technology, is offered as a means to bring software security issues under

Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop (SEW'03)
0-7695-2064-2/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

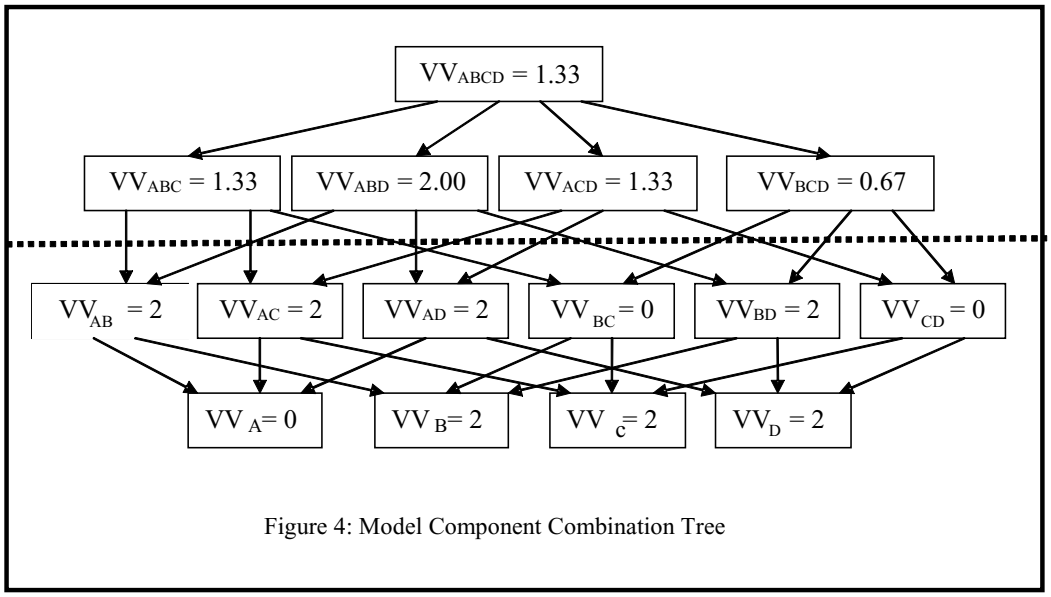

Figure 4: Model Component Combination Tree

formal control early in the life cycle. [3,4] The FMF seeks to address the problem of formal verification of larger systems by a divide and conquer approach. [5] It accomplishes this by verifying a property over portions of the system, then incrementally inferring the results over larger subsets of the entire system. As such, the FMF is: 1) a system for building models in a component based manner to cope with system evolution over time and, 2) an approach of compositional verification to delay the effects of state space explosion. This methodology allows property verification results of large and complex models to be examined and extrapolated appropriately. (See Figure 4)

Modeling in a component-based manner involves building a series of small models, which later will be strategically combined for system verification purposes. This strategic combination correlates the modeling function with modern software engineering and architecture practices whereby a system is divided into major parts, and subsequently into smaller detailed parts, and then integrated to build up a software system. An initial series of simple components can be built when few operational specifics are known about the system. However, these components can be combined and verified for consistency with properties of interest such as software security properties.

The approach of compositional verification used in the FMF seeks to verify properties over individual model components and then over strategic combinations of them. The goals of this approach are to: 1) infer verification results over systems that are otherwise too large and complex for MC from results of strategic subsets (combinations) while minimizing false reports of defects; 2) retain verification results from individual components and component combinations to increase the efficiency of subsequent verification attempts in light of modifications to a component.

FMF is being used to verify a portion of certificate handling of the Secure Sockets Layer (SSL) protocol as a demonstration of the modeling framework. SSL is designed for securing network communications. It provides the capability for authenticating communicating partners.

5. Property-Based Testing

Property-based testing is a testing technique designed to detect violations of given properties. In this context, the properties desired are obtained from the checklist, or from the properties used in the model checking. The properties are viewed as invariants that are to hold as the program executes. PBT views the execution of the program as a sequence of state transitions. If any of these transitions cause a violation of the properties, PBT causes an error message to report the failure (see Figure 5 below).

First, the properties are expressed in a low-level testing language called TASPEC. A tool called the instrumenter takes these properties and a program to be tested. The instrumenter then

Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop (SEW'03)
0-7695-2064-2/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

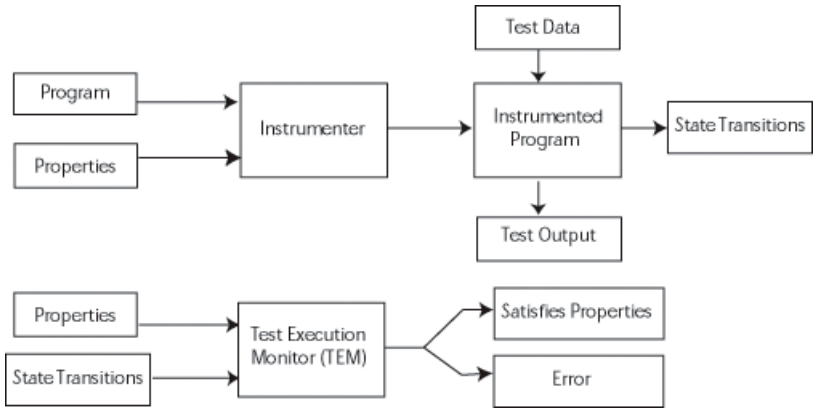

Figure 5: PBT Process

modifies the program so that, when any event occurs that affects whether the state of the program's execution satisfies the properties, a text representation of the change in state is emitted. The instrumented program is then compiled and executed. After the execution is complete, the changes of state will have been saved to a file.

The testers then execute a second program called the test execution monitor (TEM). This program is given the properties (in TASPEC) and the changes of state generated from the test run of the program. The TEM then checks each change of state to ensure that, if the properties held when the program began execution, then they held throughout the execution. If not, the TEM can determine where in the program the failure occurred.

PBT is different than formal verification. It recognizes that implementation difficulties, and environment considerations, may affect conformance to the properties (and hence the security of execution). A key observation is that testing does not validate that a program will always meet the properties, unless all possible paths of execution are traversed. But it does provide additional assurance that the implementation is correct, and does satisfy the properties, when execution follows the tested control and data flow paths.

Many control and data flow paths are irrelevant to the program's satisfying the desired properties. A technique called *slicing* [6] creates a second program that satisfies the properties if, and only if, the original program satisfies those properties. The second program contains only those paths of control and data flow that affect the properties. This focuses the testing on paths of execution relevant to the security properties, rather than on all possible paths of execution (See Figure 6).

The property-based tester currently handles Java programs, and has found vulnerabilities in several programs, including a server of several thousand lines. The instrumenter is language dependent because it must parse the program to be tested in order to add the appropriate code to print the relevant changes of state. The current instrumenter handles Java, and one currently is being written for C.

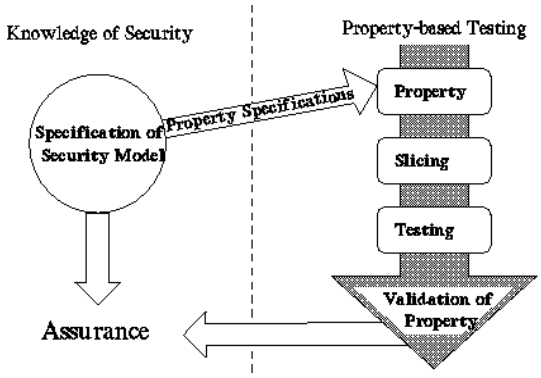

Figure 6: PBT Model

6. Software Security Checklist (SSC)

The SSC has two foci: 1) a checklist for software developers to write secure code for applications (including tools to integrate security into the various stages of the software life cycle); and 2) a checklist to verify that software released by NASA does not allow unauthenticated access into NASA networks, or provide other information about NASA's, processes, systems, networks, or other sensitive data (such as IP Address space, HR data, or processes that can be exploited).

A checklist for the development and maintenance life cycles begins with system

Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop (SEW'03)
0-7695-2064-2/03 $ 17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

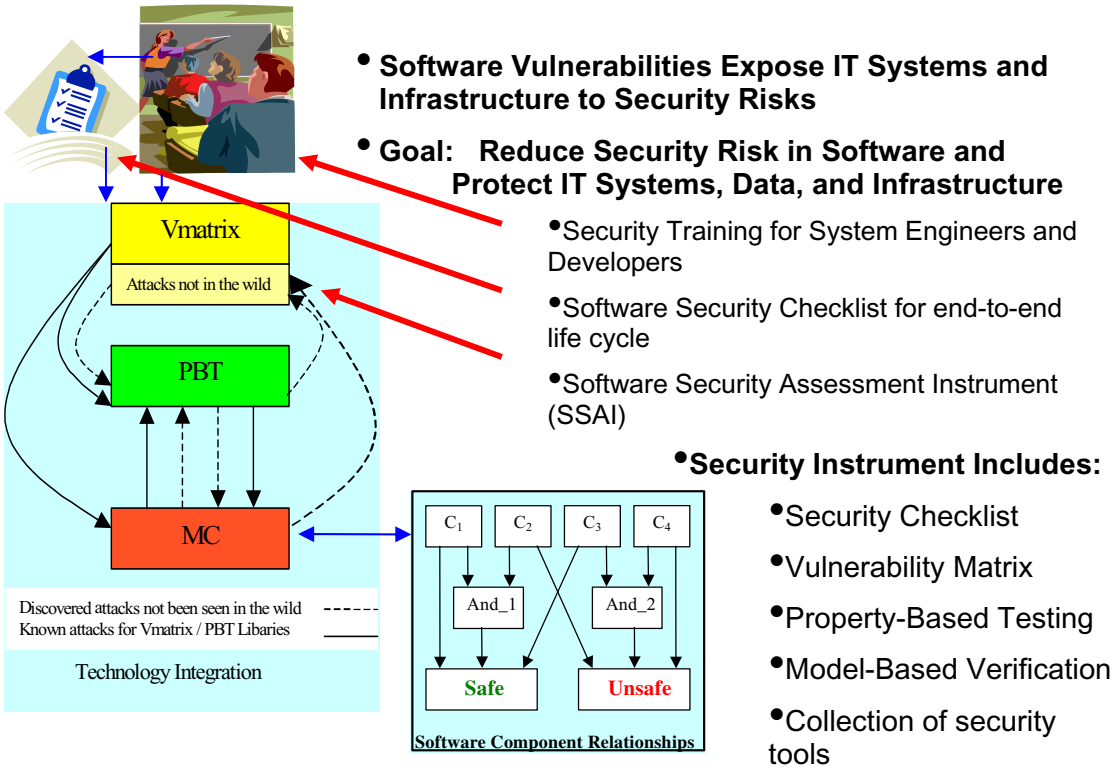

Figure 7: Integrated Use of the Security Assessment Instrument

inception and continues through retirement known as a 'womb to tomb' process. It begins with a pre-requirements study to be able to elicit the appropriate requirements from the stakeholders and applicable documents, standards, regulations, laws, etc. and specify them; and it ends with decommissioning software and systems and the impact on the computing environment, including re-verification of systems from which critical software has been decommissioned. In between there are a number of critical design and programming issues as well as tracing security requirements, and performing test and verification of them, including the maintenance life cycle phase.

A checklist for the external release of software (i.e., software that is developed for release external to the organizational environment) is specified by NASA in NPG 2210, NASA Procedures and Guidelines: External Release of NASA Software."[7] However, there is no guidance provided on the contents of the checklist nor a release authority process. The research initiative delivered to NASA a draft of a potential checklist and release authority process to be used for the external release of software. A process for evaluation of code for potential security issues was also provided. The evaluation of the source code included looking for problems that might expose NASA and NASA partners to potential security exposures. Included in the release checklist is a sample set of PERL scripts to aid in looking for potential items in the software that may violate security requirements or present security risks such as embedded Center IP addresses, Human Resource information like phone numbers, use of known vulnerable libraries, and weak subroutines.

7. Software Security Assessment Instrument (SSAI)

This collection of tools and utilities, collectively named the Software Security Assessment Instrument (SSAI), can be used individually or in concert to ensure the security of network aware application software and systems as shown in Figure 7 below. Working together, the various tools and utilities provide a distinct advantage whereby each tool's output may be used for input for the other tools. The use of these tools and instruments results in a

Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop (SEW'03)
0-7695-2064-2/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

more comprehensive assessment of the software undergoing analysis.

In response to NPG 7120.5B, "NASA Program and Project Management Processes and Requirements," [8] and NPG 2810.1, "Security of Information Technology," [9] an effort is underway to integrate security into JPL's project life cycle. The SSAI along with experts in the use of these instruments and tools will be made available to projects as part of a security risk assessment for the project life cycle.

In addition, the instrument will be evaluated for use in the Deep Space Mission Systems (DSMS). An evaluation of the applications and protocols used to support flight and ground support systems will be assessed for further piloting of the SSAI and integrating the technology into this environment.

This instrument also has the potential for application to domain areas beyond software security such as software risk management. Exploration of extension of the instrument is also being explored.

Training courses for project managers and software developers are currently being developed to integrate security and elements of the SSAI into the project life cycle. In addition, an approach is being pursued to have available at the various NASA Centers domain experts to assist projects with software and system security and methods to identify and mitigate risk in the project life cycle. It is hoped through these efforts that NASA will be able to produce software that has a higher level of assurance that security defects and unwanted exposures are not present in the final product.

8. Conclusion

A unified approach to software security has the potential to identify software security weaknesses. An approach that addresses security issues early in the life cycle increases that potential. Unifying the model-based approach with property-based testing through the use of temporal logic properties provides consistent verification across life cycle phases. The VMatrix provides the basis for instantiating the temporal logic properties for both MBV and PBT. Other security assessment tools like fault injection and fault trees can be used cooperatively or independently to address other software security concerns. The software security checklist identifies the critical areas in security that need to be addressed in the life cycle. The outcome is an integrated approach to reducing software security risk.

9. Acknowledgements

The research described in this paper is being carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

10. References

[1] C. Mann. "Why Software Is so Bad," Technology Review (July/August 2002).

[2] G. Holzmann. Design and Validation of Computer Protocols. Prentice Hall 1990; ISBN: 0135399254

[3] D. Gilliam, J. Kelly, J. Powell, M. Bishop. "Development of a Software Security Assessment Instrument to Reduce Software Security Risk" Proceedings of the Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Boston, MA, pp 144-149.

[4] D. Gilliam, J. Powell, J. Kelly, M. Bishop. "Reducing Software Security Risk Through an Integrated Approach", IEEE Goddard 26th Annual Software Engineering Workshop.

[5] Component Based Model Checking, J. Powell, D. Gilliam. Proceedings of the 6th World Conference on Integrated Design and Process Technology, June 23-28, Pasadena CA, p66 & CD

[6] M. Weiser. "Program Slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352–357 (July 1984).

[7] NPG 2210. "NASA Procedures and Guidelines: External Release of NASA Software," January, 2002, August 26, 1999, http://nodis3.gsfc.nasa.gov/library/lib_docs.cfm?range=2

[8] NPG 7120.5B. "NASA Procedures and Guidelines: NASA Program and Project Management Processes and Requirements," November, 2002, http://nodis3.gsfc.nasa.gov/npg_img/N_PG_7120_005B_/N_PG_7120_005B_.pdf

[9] NPG 2810.1. "NASA Procedures and Guidelines: Security of Information Technology," August, 1999, http://nodis3.gsfc.nasa.gov/library/displayDir.cfm?Internal_ID=N_PG_2810_0001_&page_name=main

IEEE COMPUTER SOCIETY