Reducing Software Security Risk Through an Integrated Approach

David P. Gilliam
Caltech, Jet Propulsion Laboratory
david.p.Gilliam@jpl.nasa.gov

John C. Kelly
Caltech, Jet Propulsion Laboratory
john.c.kellyw@jpl.nasa.gov

Matt Bishop
University of California at Davis
bishop@cs.ucdavis.edu

Abstract

This paper discusses new joint work by the California Institute of Technology s Jet Propulsion Laboratory and the University of California at Davis sponsored by the National Aeronautics and Space Administration to develop a security assessment instrument for the software development and maintenance life cycle. The assessment instrument is a collection of tools and procedures to support development of secure software.

The toolset initially will have a Vulnerability Matrix (VMatrix) with severity, frequency, platform/ application, and signature fields in a database keyed on the Computer Vulnerability Enumeration (CVE) number. The toolset also will include a property-based testing tool to slice software code looking for specific vulnerabilities using signatures from the VMatrix. A third component of the research underlying this toolset will be an investigation into the verification of software designs for compliance to security properties. This is based on model checking approaches initially researched together with analytical verification of formal specification.

Keywords

Security Toolset, Vulnerability Matrix, Property-Based Testing, Model Specification Checking, Security Verification

1. Introduction

Security vulnerabilities in software on networked systems provide attackers an avenue to penetrate those systems. The source of these security weaknesses are usually traced to poor software development practices, non-secure links between computing systems and applications, and mis-configurations. An otherwise secure system can be compromised easily if a system or application software on it, or on a linked system, has vulnerabilities.

Currently, there is a lack of security assessment tools for use in the software development and maintenance life cycle to mitigate these vulnerabilities. The National Aeronautics and Space Administration (NASA) has funded the Jet Propulsion Lab in conjunction with the University of California at Davis (UC Davis) to begin work on developing a software security assessment instrument for use in the software development and maintenance life cycle.

The goal of this work is to use a formal analytical approach to integrate security into existing and emerging techniques for developing high quality software and computer systems. The approach will be multifaceted, with activities and prototype tools in the following sub-domains:

– Assessment instrument for reducing risk during development, configuration, and installation of secure systems
– Models based development and verification for secure software architectures
– Security testing, and verification and validation (V&V) techniques

Assessments of high profile NASA systems believed to be vulnerable to attack will provide a metric to

determine the effectiveness of these activities and prototypes.

2. Securing the Computing Environment

The computing environment upon which this work is based consists of multiple computer systems connected by a network and running or supporting network-aware applications. A vulnerability on any one of the systems puts all the systems sharing the computing environment at risk (due to the notion of the weakest link, or, more formally, the principle of least common mechanism [A]). It is imperative to secure the computing environment to ensure NASA's computers, and hence missions and research, function as expected. One key element of securing the environment is knowledge of the systems, the software running on them and their potential vulnerabilities. Unfortunately, new exploits are being discovered daily. Consequently, maintaining the security of a computing environment is a constantly moving target that consumes institutional resources.

Developers and maintainers of software need an integrated approach to prevent security vulnerabilities from being introduced during the software development and maintenance life cycles. Much like preventing application fault errors, preventing conditions that allow the code to be exploited through race conditions, buffer overflows and the like, requires an extendable and modifiable toolset to assist developers to write code not containing these problems.

As stated in the National Institute of Standards Technology (NIST) handbook on computer security, "Security, like other aspects of a computer system, is best managed if planned throughout the computer system life cycle." ([2], p. 74) This life cycle includes planning and implementing software security. It is much less costly to plan and implement software security from the beginning of the development effort than to implement and add it later. It also ensures that security is included in the ongoing software life cycle development, including maintenance, system upgrades and the design of new modules. Life cycle management also helps document security-relevant decisions, in addition to assuring management that security is fully considered in all phases. Security management should begin early in the software life cycle and continue throughout, including during the maintenance and upgrade phases.

Our approach is to develop and implement a software security assessment instrument that can be used in the software development and maintenance life cycle to reduce risk through an integrated approach.

The assessment instrument under current development includes a Vulnerability Matrix (VMatrix), a property-based testing tool, and model-based security specification and verification mechanisms.

3. Vulnerability Matrix

The VMatrix is a dataset ranking severity of vulnerabilities against frequency of occurrence by platform. The matrix suggests where to best expend effort in minimizing security risks in the computing environment.

Its purpose is twofold: 1) to provide system administrators metrics that show where best to expend their efforts in protecting their computing environments; and 2) to provide vulnerability specifications for the property-based testing tool, enabling developers to test for vulnerabilities in code during the software development and maintenance life cycle.

The VMatrix will be stored in a SQL searchable database so its users can search for particular vulnerabilities and obtain different views of the data. The database can also be searched by platform, application, or vulnerability, so users can examine existing tools and systems as they develop new ones, or bring on line entities described in the database.

An effort to provide a unique identifier to each vulnerability is currently underway, headed by MITRE and the SANS Institute. Numbered vulnerabilities are stored in a Common Vulnerabilities and Exposures (CVE) list. A number of companies selling security products use the CVE numbers to identify vulnerabilities.

The VMatrix will use the CVE assigned identifiers as the primary key in the database. The fields will include the CVE identifier, vulnerability name, type, platform(s), application, severity, frequency, and vulnerability signatures, as well as preconditions for the vulnerability to arise. The vulnerability signatures will have multiple subfields to break up the signature into specific components. This allows the database to contain fine-grained details of the attacks to exploit the vulnerability.

The signature fields will provide developers the ability to import the signature(s) into intrusion detection tools. From those signatures, developers can derive low-level specifications to give to the property-based testing tool. This tool will use those specifications to test software for specific vulnerabilities.

4. Property-Based Testing

The role of property-based testing is to bridge the gap between formal verification and *ad hoc* verification. This provides a basis for analyzing software without sacrificing usefulness for rigor, yet capturing the essential ideas of formal verification. It also allows a security model to guide the testing for security problems.

Property-based testing [4] is a technique for testing that programs meet given specifications. The tester gives the specifications in a language that ties the specification to particular segments of code. The specification has *assertions*, which indicate changes in the security state of the program, and *properties*, which describe a specific

security state (that, in this context, is considered secure). The idea is to ensure that the properties always hold.

To simplify the testing procedure, the program is *sliced* to delete those parts of the program unrelated to the properties being tested. The result of the slicing is a compilable program that satisfies the properties if, and only if, the unsliced program satisfies the properties.

The sliced program is then modified to function as a simple state machine by inserting code to emit information about assertions and the need to check properties. A monitor collects this information and updates the state of the program. When a property statement is reached, the monitor determines if the property holds. If not, the monitor reports that the program failed to meet its specification, and the tester can take appropriate action.

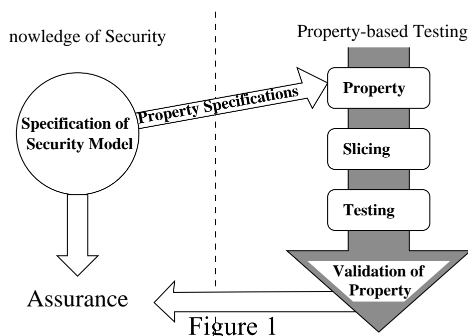

Figure 1

Figure 1 above graphically portrays how the procedure works. The circle on the left represents an accurate specification of the security model (which says what you are, and are not, allowed to do). The goal is to analyze the software to determine the level of assurance, or belief that the program does what it is supposed to do. The arrow going from the left to the right shows that the security property specifications, written in a low-level specification language called TASPEC, represent the requirements of the security model. The arrow goes to the property box on the right to emphasize that the properties we test for are taken directly from the security properties of the model. It is expected, and must be tested, that the code will honor these properties.

Next, the program is sliced, or the smallest program equivalent to the original with respect to the stated properties is derived. Then the program is instrumented and tested, as described earlier. The testing either validates the properties or shows they do not hold. This helps determine the level of assurance of the program, as captured by the arrow going from the right to the left.

As an example, consider Fink's implementation of property-based testing. He defined a language called TASPEC that expressed specifications in terms of C code (which was the environment in which his tool was to be used). This differs from more familiar specification languages such as Z [B], which work at a more abstract level. For example, consider the high-level requirement (stated in English) that "a user must authenticate himself or herself before acquiring privileges." The low-level specification (again in English) for a UNIX program written in C would be:

```
Is password corrrect? {
        Compare user's password hash to hash stored
                for that user name
        If match, set UID to user's uid
        If no match, set UID to ERROR
}
if privileges granted {
        compare UID to the uid for which privileges
                are granted
        if match, all is well
        if no match, specification violated
}
```

Translating this into TASPEC gives [3]:

location func *setuid*(*uid*) **result** 1
    { **assert** *privileges_acquired*(*uid*); }
location func *crypt*(*password*,*salt*) **result** *encryptpwd*
    { **assert** *password_entered*(*encryptpwd*); }
location func *getpwnam*(*name*) **result** *pwent*
    { **assert** *user_password*(*name*,
        pwent->*pw_passwd*, *pwent*->*pw_uid*); }
location func *strcmp*(*s1*, *s2*) **result** 0
    { **assert** *equals*(*s1*, *s2*); }
password_entered(*pwd1*) **and**
    user_password(*name*, *pwd2*, *uid*) **and**
        equal(*pwd1*, *pwd2*)
    { **assert** *authenticated*(*uid*) ; }
authenticated(*uid*) **before** *privileges_acquired*(*uid*)

The set of statements with **assert** in their associated blocks are the set of TASPEC statements that indicate changes of state. From the top, they say that when the program makes a *setuid* system call with an argument of *uid*, the process acquires the privileges of that *uid*; when the process calls the UNIX function *crypt*, the result is a hashed password; when the process calls the *getpwnam* function, it gets back information about the user, her (hashed) password, and UID; and the UNIX function *strcmp*, when called, compares two strings and returns 0 if they are equal. Code is added to the named functions so that, for example, when *setuid* is called, the assertion *privileges_acquired*(*uid*) is added to the current security state. When all of *password_entered*(*pwd1*), *user_password*(*name*, *pwd2*, *uid*), and *equal*(*pwd1*, *pwd2*) are in the state, the monitor will add *authenticated*(*uid*) to the state.

The last line says that when the assertion *setuid*(*uid*) is added to the state, the property that *authenticated*(*uid*) is already in the state, for the same *uid*, must hold; in other words, if *authenticated*(*uid*) is not in the current security state, the monitor will raise a warning that a property has not been satisfied.

This example shows how property-based testing can take advantage of the specifications of vulnerabilities in VMatrix to detect problems. The key is to represent the vulnerability in the low-level specification language. That explains the subfields in the matrix; they must capture the essence of the problem. As a side benefit, this work feeds directly into the Davis model for vulnerabilities [C], because the assertions are the preconditions of that model.

A second advantage of the properties being stored in VMatrix is that they form the core of a library that can be used for testing the security of programs other than those in the NASA suite. In essence, we extend the notion of software reuse to properties and assertions [D].

We will apply the property-based testing tool to the NASA environment. Hence, the prototype property-based testing tool will be written for C++ and the UNIX environment. Success will be determined by performing static analysis of several programs of interest. Flaws not known to those performing the testing will be injected into the program to provide a baseline of measurement. Once this is completed, we will develop and prototype an engine to perform testing on programs in a production type environment.

At this point we will examine the use of a run-time testing tool similar to the static property-based testing tool (the difference being the omission of slicing).We will determine how much its use degrades performance, and how much extra overhead the testing adds to the system in general. We will also perform friendly penetration attacks on the relevant software, and determine if the run-time testing environment detects these attacks .

The property-based software tool, Tester's Assistant—Next Generation (TANG),will benefit users by giving them increased confidence in the programs' correctness with respect to the stated (security) specifications. If non-security properties (such as safety) are of interest, the testers will have access to TANG to perform similar testing for their own set of properties. But the focus of this study is on security-related properties.

5. Model-Based Security Specification and Verification

Analyses based on models can be used to verify and check compliance to desired security properties. Many security properties cannot be verified by test activity alone, however verification through analyses and modeling at the design stage can increase the confidence that the specification provides a sound base for developing a secure program or communication protocol. The analysis and modeling process can begin early in the software development life cycle and provides a machine-readable model, which can be probed through various tools. Analyses and models should be updated periodically, as requirements and designs become more mature. Analyses and models can contribute to the verification by testing programming code through consistent collaboration of test logs.

Software model checkers automatically explore all paths from a start state in a computational tree. The computational tree will contain repeated copies of subtrees. The objective is to verify models over as many scenarios as feasible. Since the models are selective representation of functional capabilities under analysis, the number of feasible scenarios are much larger than the set that can be checked during testing. Model Checker differ from traditional formal techniques by the following characteristics:

- Model checkers are operational as opposed to deductive
- Model checkers provide error traces
- Their goal is oriented toward find errors as opposed to proving correctness (since complete state space exploration is usually infeasible)

Model based security specification and verification:

Model checking addresses issues in security protocols by examining a large number of ways to circumvent the security mechanism. In contrast to purely analytic methods, model checking is capable of examining the larger venue by validation of the over-all-security system in local, regional, or global environments. These methods have more leverage since they model real world scenarios, and they embrace more than just the mathematics of the protocol. For example, the Needham-Schroder protocol (1978) was proven secure using the BAN logic for protocol specification. However, Lowe (1998) and Wu (1998) using the model checking system SPIN, have discovered successful attacks abrogating the effectiveness and usefulness of this protocol. We propose to extend this approach to protocol validation by (1) proposing models of security protocol systems, and (2) validating those configurations. These modeling techniques have developed around a multi-agent programming paradigm that has emerged as a convenient framework around which internet applications can be successfully validated Mitzoguchi(1998).

Consider two concurrent processes P1 and P2 depicted by the following state machine diagrams (example adapted from Callahan*)

y

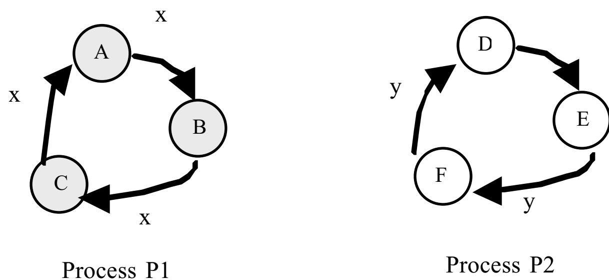

Figure 2

A partial state transition diagram for the joint process machine would be:

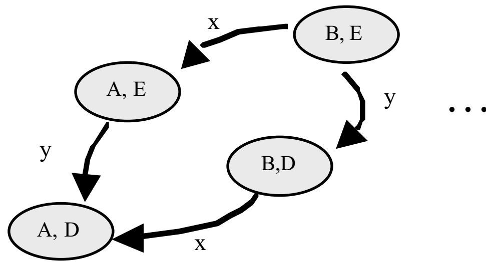

Figure 3

Model checkers effectively and automatically explore a large number of paths from a start state in a computational tree. In Figure 4 below we have diagramed the paths of the model checking. Note we have not listed in the tree any state that was reached in a prior level. The computational tree will contain repeated (perhaps infinitely many times) copies of subtrees. We ve arbitrarily chosen state (A,D) as the start state for this computational tree.

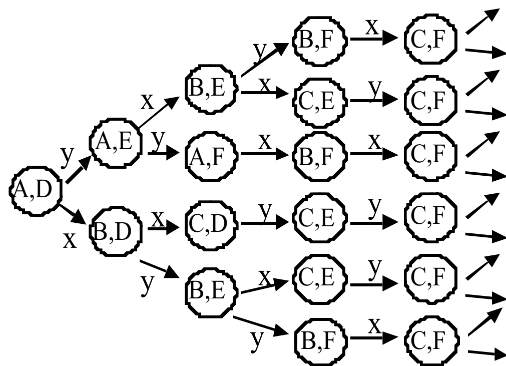

Figure 4

Considered together, the joint process machine has nine states (the Cartesian product of the state space for P1 with the state space for P2):

(A,D)	(B,D)	(C,D)
(A,E)	(B,E)	(C,E)
(A,F)	(B,F)	(C,F)

State Space for Joint Process Machine*

(* Examples is from a presentation by J. Callahan entitled *Automated Testing via Model Checking*, West Virginia University, 1997)

Figure 5

Three common properties to check for are:

Invariant — always p

- p is a property the model must always have

Safety — not ever q

- q is a property the model must never have

Liveness — r implies s will be true now or in the future

- always the case that if property r holds at the current state, then property s will hold at some state now or in the future
- used to guarantee that significant sequences take place

Security verification of new architectures:

Architectures that support change and facilitate maintenance are essential to secure systems. However, these architectures are inadequately tested by traditional verification techniques. Model Checking offers ways to begin modeling and investigating the behavior of the planned system, and to validate that key properties hold invariantly in the system as modeled. This technique will be explored in collaboration with security vulnerabilities and property based testing as part of this study.

6. Conclusion

The three parts of this work form a coherent technique to examine new and existing systems for security flaws. The vulnerability matrix drives the selection of security properties that the Tester's Assistant — Next Generation will look for. The matrix contains the problems of greatest concern to NASA at the moment. Model checking will refine the selection of properties by moving their selection from an ad hoc technique to an analytic technique based upon the needs of the systems. So, the Vulnerability Matrix and model-pased checking provide the properties that the software must meet; the property-based tester checks that the programs do indeed meet these properties.

This project suggests several other area in which fruitful research may be conducted.

* The model-checking methodology requires a secure model to check against. ˚ Developing such models is an open research issue.
* Property-based testing requires properties expressed in TASPEC to test ˚ against. The vulnerability matrix forms the beginning of a library ˚ of TASPEC properties. How to develop other properties worth adding to ˚ a library of properties is another open research issue.

* Training in the writing of more secure programs flows directly from the ° library of security properties and the system-specific models. Placing ° these in the context of a particular language and environment is an ° important part of improving the quality of software and systems.

Porting the toolset to alternate environments (such as Windows 2000) and ° languages (such as scripting languages and Java) requires revisiting the ° models used to develop the tools as well as the models of systems.

7. Acknowledgements

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

The research described in this paper is also being carried out by the University of California at Davis under a subcontract with the Jet Propulsion Laboratory, California Institute of Technology.

8. References

[1] M. Burrows, M. Adbadi, and R. Needham. A Logic of Authentication. Technical Report 39, DEC Systems Research Center, February 1989.

[2] An Introduction to Computer Security: The NIST Handbook," National Institute of Standards and Technology Administration, U.S. Department of Commerce, 1995.

[3] G. Fink, M. Bishop, Property Based Testing: A New Approach to Testing for Assurance, *ACM SIGSOFT Software Engineering Notes* **22**(4) (July 1997).

[4] G. Lowe. Breaking and Fixing the Needham-Schroeder Public Key Protocol Using CSP and FDR. In TACAS96, 1996.

[5] F. Mizoguchi, H. Ohwada, H. Nishiyana, and H. Hirasishi. An Integrated Agent Architecture for Smart Office Robot Collaboration. Technical report, Information Media Center, Science University of Tokyo, 1998

[6] "Panel I: Issues in Requirements Definition for Survivable Systems," Chair: R. C. Linger, Proceedings of the 3rd International Conference on Requirements Engineering (ICRE '98), Apr, 1998, Colorado Springs, CO, pp. 198-199.

[7] A.B. Smith, C.D. Jones, and E.F. Roberts, Article Title , *Journal*, Publisher, Location, Date, pp. 1-10.

[8] W. Wen and F Mizoguchi. Model checking Security Protocols: A Case Study Using SPIN, IMC Technical Report, November, 1998.

[A] J. Saltzer and M. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE* **63**(9) (1975) pp. 1278-1308.

[B] A. Diller, *Z: An Introduction to Formal Methods*, John Wiley and Sons, New York, NY (1990).

[C] M. Bishop, "Vulnerabilities Analysis," *Proceedings of the Recent Advances in Intrusion Detection* (Sep. 1999).

[D] J. Dodson, "Specification and Classification of Generic Security Flaws for the Tester s Assistant Library," M.S. Thesis, Department of Computer Science, University of California at Davis, Davis CA (June 1996).