Application of Lightweight Formal Methods to Software Security

David P. Gilliam,* John D. Powell,* and Matt Bishop**
*Jet Propulsion Laboratory, California Institute of Technology
** University of California at Davis
{dpg, john.d.powell}@jpl.nasa.gov, bishop@cs.ucdavis.edu

Abstract

Formal specification and verification of security has proven a challenging task. There is no single method that has proven feasible. Instead, an integrated approach which combines several formal techniques can increase the confidence in the verification of software security properties. Such an approach which specifies security properties in a library that can be re-used by 2 instruments and their methodologies developed for the National Aeronautics and Space Administration (NASA) at the Jet Propulsion Laboratory (JPL) are described herein The Flexible Modeling Framework (FMF) is a model based verification instrument that uses Promela and the SPIN model checker. The Property Based Tester (PBT) uses TASPEC and a Test Execution Monitor (TEM). They are used to reduce vulnerabilities and unwanted exposures in software during the development and maintenance life cycles. These instruments are currently being piloted with a COTS Server-Agent Application.

1. Introduction

Specifying software properties is a challenging task. Even more challenging is specifying informal specifications formally. [1] This difficulty is due to the imprecision of natural language and the difficulty in ensuring that the specifications are correct. [2, 3] Applied to security, formal specification is particularly complex as security requirements mostly state what must not happen. [4] The problem of trying to specify security properties formally was made apparent during the 1970's when the United States government commissioned development of a provably secure multics system using mathematical modeling. [5] Their approach addressed only confidentiality, and then only partially. The number of follow-on discussions on security property specifications is witness to this problem. [4, 6, 7, 8] The need to formally specify and verify security properties is easily seen by the growing list of software vulnerabilities. [9] It is apparent that better specification and verification of security properties will lead to more secure software. Formal specifications and methods can fill this role and improve the quality of software making it more dependable. [8, 10]

The following discussion will focus on 2 formal, integrated techniques, model checking and property-based testing, that are being used at the Jet Propulsion Laboratory (JPL) for verification of security properties. They are being piloted with a Commercial-Off-The Shelf (COTS) application that has a Server-Agent function, where the Server provides application software and updates to agents running on associated workstations. The agents check in with the server and download applications. The process requires verification that the agents do not provide a source of vulnerabilities or exposures to the systems in their operation.

2. Model Checking and Property-Based Testing

The purpose and use of tools like model checkers and testers is to allow for mechanization of formal specifications to reduce cost and schedule while increasing efficiency for formal verification activities and to assure that the software artifacts are free from potential conflicts and violations in the specifications. [11, 12] Model checking involves:

- Building a state-based model of the system
- Identifying properties to be verified
- Checking the model for violations of the specified properties.

Model checkers such as SPIN [18] automate the process of verifying a property over its corresponding model.

Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE'05)
0-7695-2362-5/05 $20.00 © 2005 IEEE

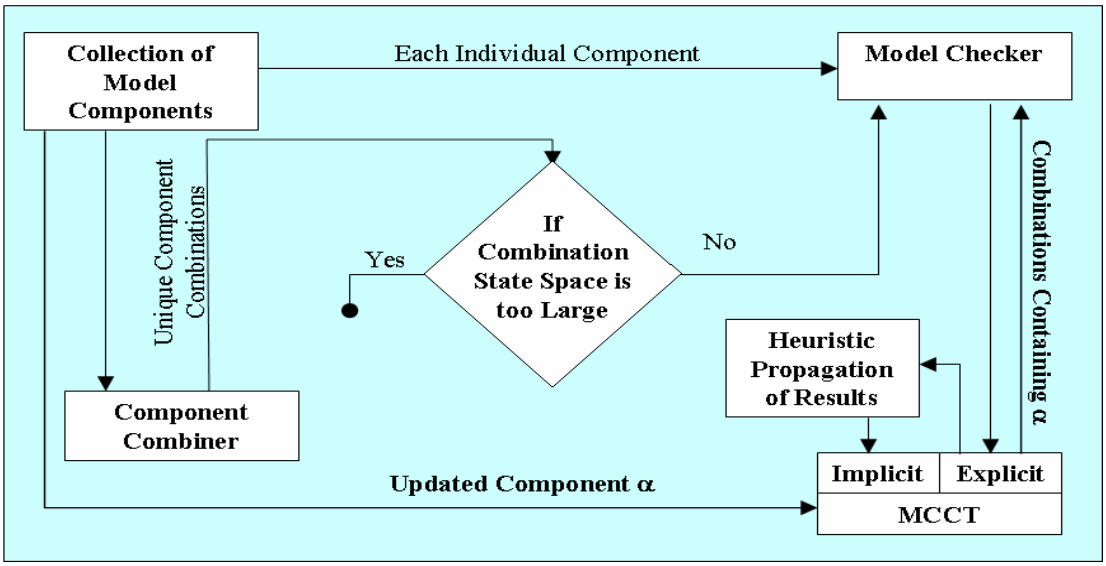

Figure 1: The Flexible Modeling Framework And Combiner Process [15]

Model checkers perform an exhaustive search of a state space generated by a model. State space is the set of total reachable system states represented in the model. A given state consists of all variables in the model and their associated values at a given point in time. Software model checkers automatically explore all paths from a start state by examining transitions in the state space to determine the reachability of a state that violates a property. [8] The properties are verified as holding or not holding for each transition. This automation provides high value for large, complex systems where specifications are complex. [13]

As the size and complexity of the model increases, the state space to be checked grows at an exponential rate. "This exponential growth in the state space known as the state explosion problem is the limiting factor in applying automatic verification methodologies to large systems." [14]

2.1. Flexible Modeling Framework

To address the state explosion problem, JPL has developed a Flexible Modeling Framework (FMF) that uses a "divide and conquer approach" while seeking to maintain fidelity to the software artifact. [15] The FMF uses compositional verification to analyze models and verify the results for models that represent the system. The basis of the compositional approach is the verification of a system with regard to a subset of its environment in a manner that allows those results to be extrapolated to the environment at large. The FMF approach narrows the focus to those components for which security properties have been identified and which can be modeled. [16]

As previously reported to WETICE, use of this combinatorial approach allows interactions between components to be examined bringing to light potential questions about their relationships.[16] These questions enable decisions to be made early in the life cycle. Further, efficient, localized updates of the system model can more easily be generated. Issues affected by subsequent changes can then be revisited though required re-verification of affected combinations. This last feature is a failsafe and not a substitute for good practices such as documentation of decisions and emergent requirements.

The FMF takes the most critical software components and builds models of them. Security properties are then verified in each component. The interacting components are then combined and model checked for violations of properties. This approach allows more of a system to be model-checked within system resource constraints thus providing a higher degree of confidence (Figures 1 and 2). The concept is to: a) verify systems that are otherwise too large and complex by checking only strategic components and "b) retain verification results from individual components and component combinations to increase the efficiency of subsequent verification attempts in light of modifications to a component." [16]

The model component combination tree in Figure 2 shows the combination of components that interact with each other. The components that interact are combined and model-checked within the FMF. The paths from the higher level components to the lover level components that interact are shown in the model.

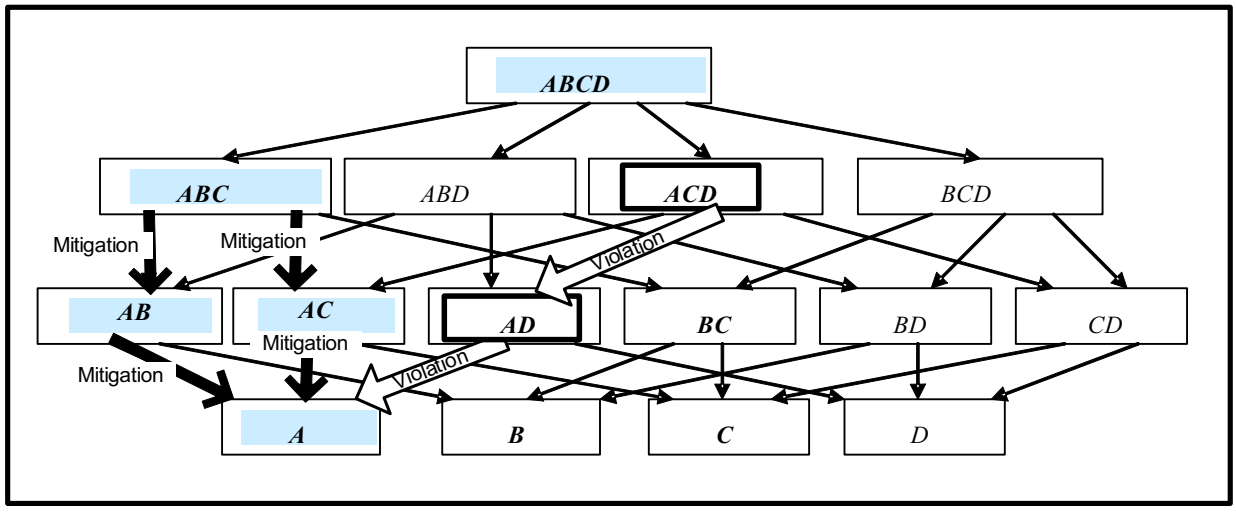

Figure 2: Model Component Combination Tree and Components Model Checked [19]

As an example, suppose in the Model Component Tree (Figure 2) the path from *ACD* to *AD* to *A* would normally produce a violation in the model verification. However, in this case the path from *ABC* to *AB* and *AC* represent mitigations that safeguard against the violation to *A*. Consequently, the property holds at the lower level of the component tree and no violation exists unless the components *ABC*, *AB* and *AC* are removed from the tree. An example is provided in Section 3 where the MBV was used to model a COTS software product and the mitigation of a violation by two other security properties.

2.2. Property-Based Testing (PBT)

Property-based testing (PBT) is a technique that verifies that specified security properties are not violated in the coding phase of the life cycle. Properties are invariants that are to hold during program execution. Implementation difficulties and environmental considerations may affect conformance to properties (and hence the security of execution) and thus the properties may not always hold. PBT provides additional assurance that the software is correct and satisfies the specified properties when execution follows the tested control and data flow paths. [17]

A PBT instrument developed by UC Davis in cooperation with JPL, mechanizes verification of security properties in code. The PBT instrument was originally developed to check for security properties in the JAVA language. It is now being extended to check for security properties in the C language. [17]

The PBT expresses properties in a low-level test language called TASPEC. The PBT focuses the testing on the security properties of interest. Intuitively, the PBT instrument looks at the execution of program sequences as a series of state transitions. If any state transition causes a violation of a property, an error message is generated.

The PBT examines data from program executions to expose this. The goal of the PBT is to test as many paths of control as possible. First, a program called the instrumenter analyzes the security properties and the program, and inserts code to emit messages indicating changes of state relevant to the security properties. The program is then 'sliced', creating 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. The instrumented, sliced program is then compiled and executed. During execution, the messages indicating changes of state are saved to a file.

Second, a test execution monitor (TEM) program is given the properties in TASPEC and the messages indicating changes of state from the instrumented program's run. The TEM checks each state transition and verifies that the properties held during execution. If the properties did hold, then they held throughout the execution. If not, the TEM can determine where in the program the failure occurred. [17] The testing either validates the properties or shows they do not hold.

Table 1: MBV Verification Results

Agent Properties	MBV Results
1. The agent and server shall be capable of secure communication	Verified to Hold
2. The agent and server shall have an identification that uniquely mutually associates them	Verified to Hold
3. The agent and server shall authenticate to each other using their unique identification	Verified – Logically Implied by 1 and 2
4. The agent shall validate all packages that they are from its associated server	Verified – Logically Weaker version of 3
5. The agent shall validate that the package is un-tampered (like using an MD5 checksum)	Verified – Logically
6. The agent shall recognize packages that do not complete their installation	Verified to Hold - Critical
7. The agent shall have a recovery process for packages that have partial installation or otherwise fail during installation	Verified to Hold
8. The agent shall run at low priority	Verify by Other Means
9. The agent shall recognize conflicts with other processes that generate high CPU utilization	Verify by Other Means
10. The agent shall go to sleep when CPU utilization is high	Verify by Other Means
11. The agent shall monitor activity for system resources	Verify by Other Means
12. The agent shall recognize conflicts with use of JAVA resources	Verify by Other Means
13. The agent shall go to sleep when it detects conflicts with JAVA resources	Verify by Other Means
14. The agent shall only accept connections that it has initiated	Verified to Hold
15. The agent shall have a network session time-out	Verified to Hold - Critical
16. The agent shall have a package installation time-out	Verified to Hold - Critical
17. The agent shall provide logging of all its events	Verified through Inspection
18. The agent shall be capable of running as non-root and maintain reporting capabilities	Verify by Other Means

3. Prototyping the FMF and PBT

While it is believed that these verification techniques aid in ensuring that specified security properties in software are not violated, the instruments themselves must be prototyped to show that they do perform as intended and do so in a cost-effective way. Both their value and their relative cost-effectiveness must be verified for these instruments to useful and of benefit in the development and maintenance life cycles.

The MBV FMF was initially evaluated against the SSL protocol that contained a documented weakness with Man-In-The-Middle attacks coupled with a Domain Name System (DNS) spoofing attack. The MBV FMF was able to detect this weakness by encountering a path in the model of the protocol that would allow this to occur.

The PBT was evaluated initially using a JAVA based web server that had a known vulnerability. The PBT was able to detect the violation of the security property which allowed authentication to be by-passed. It reported in the code where the violation occurred and the property violated.

For further verification of the MBV FMF and the PBT, these tools are being piloted with a COTS application written in JAVA. The COTS installs or updates software from a parent server to clients. The clients communicate with the server via agents.

Since the application was already developed and available, the approach was to evaluate the purpose of the application and the properties important to its

function which have potential impact on security. The two instruments are being used cooperatively to evaluate the value of this approach. Normally, with code, it is unnecessary to model it, but to proceed directly on to property-based testing. However, the goal was to evaluate both the modeling instrument and then the ability to take the model and extrapolate from it the security properties to test.

The verification process required working with the developers of the application to obtain from them their software and architecture artifacts to extrapolate properties for the FMF and PBT. Additional information and explanation of some of the properties in the architectural artifacts was obtained from the developers. From these artifacts and in working with the developers, a model of the software was developed. Further, in view of the purpose of the application, security properties were specified independently.

A study was performed of the COTS server and agent software and its operation. It was decided that verifying the security properties of the agent software would provide the most benefit and be manageable not only to verify the security properties but also to verify the validity of the instruments and the approach.

After evaluating the software's purpose and function, properties that could impact security of the system were generated and evaluated. There were 18 essential properties that were determined as essential for the operation of the application, some of which were considered as critical security properties. These are shown in Table 1. The goal is to provide a higher level of assurance with respect to the security of the agent software through the combined the use of the FMF and PBT instruments.

The MBV verification has been completed. The FMF-style model was able to verify that the critical properties hold over the model. (See Table 1) One mitigated violation was identified. A path was discovered in which a Denial of Service could occur. The violation could be perpetrated through a constant attempt to submit packages to the agent from a source other than the parent server. The agent would continually spend time verifying and rejecting the package. However, this violation was only likely to occur when secure communication and mutual authentication were not used (properties 1 and 2). The explicit use of SSL mitigated this potential violation.

Properties 1 and 2 act as mitigation for property 4, where the system could be relegated to verifying and discarding packages that were not from the server continuously. This finding may be used in part to validate the hypothesis of the FMF that a violation in a lower level component of a model framework could be mitigated by a component or combination of components at a higher level of the model framework.

After MBV was performed, the resulting specifications of the model, the verification results, and the security properties were then passed on to the PBT for use in verification at the implementation/code level.

The PBT is testing the implementation of the tool to ensure that the properties verified by the FMF are correctly implemented. The FMF expressions of these properties are translated into TASpec. This associates the properties with the implementation of the tool being tested, and enables the PBT to record relevant changes of state during execution. Testing at the implementation level also allows us to check some properties that are not easily modeled, such as properties 12 and 13, because they identify implementation-level problems that result in changes of state—exactly the type of flaw the PBT is designed to find.

Some of the properties in Table 1 must be refined in order to encode them in TASpec. For example, the PBT can check for "secure communication" provided that the term is specified precisely and in terms of the software. If "secure communication" means using SSL, then the TASpec property would be written to ensure the SSL routines are invoked properly. (Even the SSL routines could be checked if desired.) In addition, test data must exercise the paths involved in the changes of state.

5. Conclusion

It is expected that this approach will improve the overall security of software. The results of the current investigation will be provided to the National Aeronautics and Space Administration's (NASA) Independent Verification and Validation (IV&V) Center through whom this research has been funded.

Through use of formal techniques in application to security, security assessment instruments and tools in the software development and maintenance life cycles can improve the security of software if used correctly. We hope the SSAI being developed at JPL and UC Davis is a step forward in this direction. Through the application of these instruments in a coordinated effort, a higher level of assurance for security can be achieved.

Tools and instruments that can be used during both the development and maintenance life cycle beginning with a security checklist in the inception and requirements phases through retirement will create an environment of stronger security.

7. 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.

7. References

[1] Van Lamsweerde, A., (2000). Formal Specification: A roadmap. In Finkelstein, A. (ed.) Proceedings of the Conference on the Future of Software Engineering (pp. 147 – 159). ACM Press.

2] Hussmann, H. (1997). Formal foundations for software engineering methods. Goos, G., Hartmanis, J., and van Leeuwen, J. (eds.), Lecture Notes in Computer Science, 1322. Berlin: Springer.

[3] Schach, S.J. (2005). Object-oriented and classical software engineering. 6th ed. New York: McGraw-Hill.

[4] Rushby, J. (March, 2001). Security requirements specifications: How and what? Invited Paper from Symposium on Requirements Engineering for Information Security (SREIS), Indianapolis, IN.

[5] Bell, D. E. and LaPadula, L. J. (March, 1976). Secure computer systems: Unified exposition and multics interpretation. Technical Report Mitre TR-2997, Mitre Corporation, Bedford, MA.

[6] McLean, J. (January, 1999). Twenty years of formal methods. Proceedings of the IEEE Symposium on Security and Privacy, 115 – 116.

[7] Payne, C. N., Jr., Moore, A. P., and Mihelcic, D. M. (1995). An experience modeling critical requirements. Proceedings of the Ninth Annual Conference on Computer Assurance, 1994. COMPASS '94 'Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security', 245-255.

[8] Nicol, D. M., Sandera, W. H., and Kishor, S.T. (2004). Model-Based evaluation: From dependability to security. IEEE Transactions on Dependable and Secure Computing. Vol. 1, No. 1, 48 – 65.

[9] Mitre Corporation (2004). Common Vulnerabilities and Exposures (CVE) List. Retrieved November 22, 2004, from http://www.cve.mitre.org/cve/downloads/.

[10] Easterbrook S. M., and Callahan, J. R. (1996). Formal methods for verification and validation of partial specifications: A case study. Report to NASA Independent Verification and Validation Facility, Fairmont, WV. Retrieved November 14, 2004, from www.cs.toronto.edu/~sme/papers/1998/NASA-IVV-97-010.pdf.

[11] Holzmann, G. J., & Smith, M. H. (2002). An automated verification method for distributed systems software based on model extraction. IEEE Transactions on Software Engineering, Volume: 28, No. 4, April 2002, 279 – 295.

[12] Hamon, G., de Moura, L., and Rushby, J. (May, 2004). Generating Efficient Test Sets with a model checker. Computer Science laboratory (CSL) Technical Note. SRI International. Retrieved November 14, 2004, from http://www.csl.sri.com/users/rushby/biblio.html.

[13] Rushby, J. (February, 2002). Using model checking to help discover mode confusions and other automation surprises. Reliability and System Safety. Vol. 75, No. 2, 167-177.

[14] McMillan, K. L. (1992). Symbolic model checking: An approach to the state explosion problem. CMU-CS-92-131, Submitted to Carnegie Mellon University in partial fulfillment of the degree of the requirements for the degree of Doctor of Philosophy in Computer Science. Carnegie Mellon University. Later published (1992) as Sympolic model checking. Kluwer Academic Publishers: Norwell, Mass. Retrieved Nov. 22, 2004, from www-cad.eecs.berkeley.edu/~kenmcmil/thesis.ps.

[15] Powell, J. D., and Gilliam, D. P., (2002) Component based approach to modeling for model checking. The Sixth Biennial World Conference on Integrated Design & Process Technology. Pasadena, California, 2002.

[16] Gilliam, D. P., and Powell, J. D. (2002). Integrating a flexible modeling framework (FMF) with the network security assessment instrument to reduce software security risk. Proceedings of the 11th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. WETICE 2002, 153 – 158.

[17] Gilliam, D. P., Powell, J. D., Haugh, E., and Bishop M. (2003). Addressing software security and mitigations in the life cycle. Proceedings of the 28th Annual NASA Goddard IEEE Software Engineering Workshop (SEW), 201 – 206.

[18] Holzmann, G. J. (2004). The SPIN model checker: Primer and reference manual. Boston, MA: Addison-Wesley.

[19] Powell, J. D. (2003). Reducing software security risk through an integrated approach research initiative: Model based verification of the secure socket layer (SSL) protocol. Deliverable to the NASA IV&V Facility. Fairmont, WVA.