Security Verification Techniques Applied to PatchLink COTS Software

David P. Gilliam[1], John D. Powell[1], Matt Bishop[2], Chris Andrew[3], and Sameer Jog[3]
[1]Jet Propulsion Laboratory, California Institute of Technology
[2]University of California at Davis, [3]PatchLink Corporation
{dpg, john.d.powell}@jpl.nasa.gov, bishop@cs.ucdavis.edu, {chrisa, sjog}@patchlink.com

Abstract

Verification of the security of software artifacts is a challenging task. An integrated approach that combines verification techniques can increase the confidence in the security of software artifacts. Such an approach has been developed by the Jet Propulsion Laboratory (JPL) and the University of California at Davis (UC Davis). Two security verification instruments were developed and then piloted on PatchLink's UNIX Agent, a Commercial-Off-The-Shelf (COTS) software product, to assess the value of the instruments and the approach. The two instruments are the Flexible Modeling Framework (FMF) — a model-based verification instrument (JPL), and a Property-Based Tester (UC Davis). Security properties were formally specified for the COTS artifact and then verified using these instruments. The results were then reviewed to determine the effectiveness of the approach and the security of the COTS product.

1. Introduction

Specifying software properties is a challenging task because of the imprecision of natural language and the difficulty of ensuring that the specifications are correct. [1] Better specification and verification of security properties will lead to more secure and dependable software artifacts. [2,3] JPL and UC Davis, in cooperation with PatchLink Corporation, took informal specifications, formalized them, and used model checking and property-based testing to verify the security of PatchLink's UNIX agent software.

We focus on the use of the Model-Based Verification (MBV) Flexible Modeling Framework (FMF) developed at JPL and the Property-Based Tester (PBT) developed by UC Davis to verify the security of the Commercial-Off-The Shelf (COTS) PatchLink UNIX Agent software. [4] We begin with a short discussion of the Flexible Modeling Framework and the Property-Based Tester. Next we describe the use of these two instruments with the PatchLink UNIX Agent. We present the security properties that the agent needs to satisfy, and which properties were able to be used with each instrument. We evaluate how well the instruments performed. Lastly, we summarize the results of the verification.

2. Flexible Modeling Framework (FMF) Model Checking and Property-Based Testing (PBT)

Model checkers and testers automate verification of specifications for efficiency and cost effectiveness, as well as to assure that the model of the software artifacts is free from potential conflicts and violations of the specifications. [5]

Previous publications described how the FMF and PBT instruments could be used together or independently. Used together, the instruments would confirm that security property verification results in the requirements and design phases are consistent with testing results in the coding phase of the life cycle. Used independently, the FMF and PBT can verify security properties that cannot easily be verified by the other instrument—as will be discussed below in Section 3.

2.1. Flexible Modeling Framework

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, SMV and SAL automate the process of verifying a property over its corresponding model. They require domain experts to specify the properties mathematically and then program the properties into the modeling language such as Promela for SPIN.

The FMF instrument developed at JPL uses model-based verification techniques with the SPIN

Proceedings of the 15th IEEE International Workshops on Enabling Technologies:Infrastructure for Collaborative Enterprises (WETICE'06)
0-7695-2623-3/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

model checker [6] to verify properties over a corresponding model—here security properties. As presented in a previous WETICE ST Workshop paper [7], the FMF uses a compositional approach that models interacting components and verifies security properties in each of the components and in their interactions. [Figure 1] The objective is to verify security properties for systems that are otherwise too large and complex by checking strategic components and building up a model that still maintains fidelity to the artifacts.

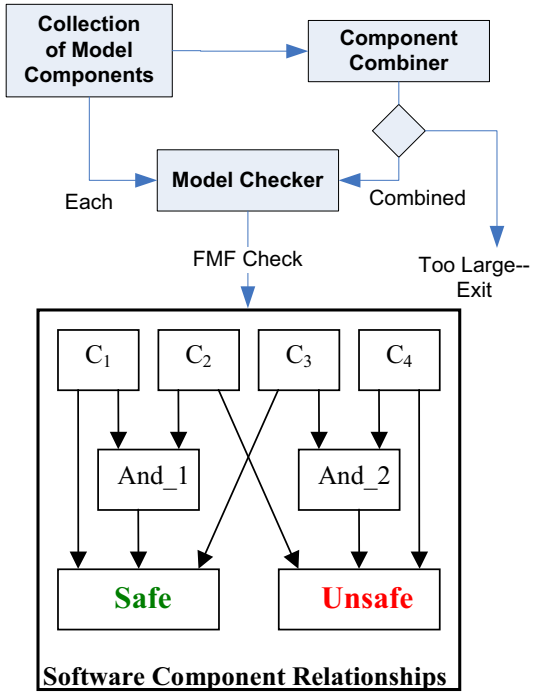

Figure 1: Flexible Modeling Framework (FMF)

2.2. Property-Based Tester (PBT)

The PBT treats paths of execution as sequences of states and state transitions. The properties describe invalid states (in which security properties are violated). The goal of the PBT is to test as many paths of execution as possible, to verify none enter an invalid state [8].

Properties are written in a low-level specification language, TASpec, that relates the property to specific code in the program.

The first step of the PBT is to analyze the software. The PBT instrumenter inserts code to print state information at locations where relevant changes of state may occur. The instrumented program is executed, and the messages emitted (called "traces") are passed to an execution monitor (TEM). The TEM also loads the specified properties, and then verifies that the properties were not violated when the program ran. If any properties are violated, the TEM identifies the violation and where in the program the violation occurred (See Figure 2). [8]

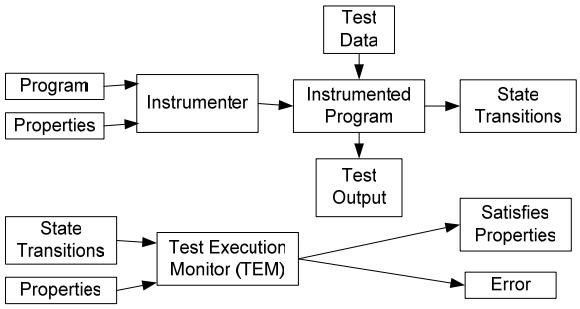

Figure 2: Property-Based Testing (PBT)

3. Related Work

There have been other efforts in both modeling and testing. The Symbolic Analysis Laboratory (SAL) from SRI International is an environment for the exploration and analysis of concurrent systems specified as transition relations [9]. The SAL toolkit provides several tools for examining SAL specifications, including three different high-performance model checkers for LTL: symbolic, bounded, and infinite-bounded. FMF allows for smaller set of interacting components to be modeled rather than the whole system, as other model checkers require. This capability helps address state-space explosion in the model while maintaining a degree of fidelity to the actual software artifacts.

Many code analyzers, such as Klocworks [10] and CodeAssure, [11] are static analysis tools that look for known vulnerabilities. In contrast, the PBT is dynamic and looks for violations of specified security properties. It can find property violations that are peculiar to the program, as well as more common vulnerabilities such as a static analyzer might uncover.

4. Verification of Security Properties

The MBF FMF and the PBT were tested on the PatchLink UNIX agent software written in Java. The goals for the test were two-fold: 1) Verify the viability of performing model checking on the design and then performing testing on the code, working from the property specifications of the model for the Model-Based Verification (MBV) instrument and the property specifications for the PBT instrument. 2) Verify the PatchLink UNIX agent for the security properties shown in Table 1. The end objective was to provide a higher level of assurance of the security

IEEE COMPUTER SOCIETY

of the UNIX agent running on the OS through the combined application of the MBV and PBT instruments to the design specifications and code.

We distinguish between modeling, which is the analysis of the design of the program, and testing, which deals with the actual software. The desired properties are the same, but modeling checks an abstraction of the actual software, whereas testing checks to validate the implementation.

The key security properties for the UNIX agent are shown in Table 1.

Table 1: Security Properties for Verification

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

4.1. Verification Activity

The security properties that were modeled were properties 1-7 and 14-16. The properties that were tested with the PBT were 5, 8, and 14. Properties 1-4, 6-7, and 15-16 were not tested for reasons provided below. Properties 9-13 depend on the system kernel to ensure that the priorities of the services are correctly handled. The agent runs at the lowest priority, and the agent lets the system determine which process is run depending on the priority level of the service request. Property 17 was observed but it is not possible to ensure that all events are logged as that would require an *n* complete path for testing. For property 18, at the time of testing, a non-root UNIX agent was not available so this could not be tested.

4.2. Model-Based Verification Summary

The results of MBV FMF prototyping activity on the PatchLink UNIX agent were previously reported at the 2005 WETICE ST Workshop [7]. Only the current effort that led to the follow-on property-based testing verification activity will be described.

The verification process required working with the developers of the UNIX agent to extrapolate properties for the MBV and PBT. From the software artifacts and design documents, a model of the software was developed and coded into Promela the language of SPIN. SPIN was then run to look for violations of the specified security properties.

Properties 8-13 and 17-18 were not able to be modeled in this effort due to the type of properties and time constraints. For instance, on property 8 continuous operations of a state cannot easily be model-checked. These properties could either best be verified using the PBT or through other means such as direct observation.

The only one minor finding was a potential for a Denial of Service (DoS) attack. Packages could be sent by a rogue server to an agent, causing it to review and reject the bad packages. If done rapidly, this prevents the agent from handling legitimate packages, and violates property 4. The violation is mitigated if secure communications are used.

4.3. Property-Based Testing Summary

The PBT tests the implementation of the software to ensure that the properties verified by the FMF are correctly implemented. It also verifies some properties not easily model-checked.

In this test, the PBT was used to verify a subset of properties due to time constraints. We used property-based testing to validate three properties for some test cases. We wrote properties and invariants

IEEE COMPUTER SOCIETY

for these, and three other properties, and tested them using various usual, unexpected, correct, and incorrect inputs. However, the testing suffered from several problems. The analyst was unfamiliar with the PatchLink source code, and was further constrained by time. As a result, some properties and invariants may be sub-optimal. The implementation of the PBT tool presented some unexpected limits, and work-arounds had to be creates. (One such limit is discussed below.) Finally the TEM is written in Prolog, and the system on which the testing was performed did not have a Prolog engine. This meant the TEM could not be used; fortunately, the traces generated by the test were simple enough to be interpreted manually.

The "goal of the PBT is to test as many paths of control as possible." [7] The focus is to test the paths of "execution relevant to the properties rather than on all possible paths of execution." The TEM verifies that the specified properties hold during execution. If they do not, the TEM announces both the violation and the location in the program where the violation occurred.

4.3.1. Property 5. [Verified]

This property requires that updates be verified by computing a CRC checksum. The checksum is computed in two places, one for ZIPped files and one for unzipped files. The following property describes the check:

```
(fileok(x,y,z) and cpfile(a,y)) or
(fileok2(x1,y1) and cpfile(a1,x1))
```

where fileok (fileok2) is true if file x (x1) checksums correctly, and cpfile is true if the file named in the first argument is copied into the file named by the second argument. The trace showed several copies of updates were not checked. Upon further inspection, most were internal copies generated by the agent. Only one was generated when the update was downloaded, and that was checked. It showed the invariant was satisfied. Only time constraints prevented refining the properties to eliminate the false positives. As a check, the source code was altered to cause an incorrect checksum to be computed. The resulting trace file showed that the invariant was no longer satisfied.

4.3.2. Property 8. [Verified]

Property 8 requires that the agent be run at a lower priority. This is done in the script "detect", which is a shell script. The instrumenter does not work on shell code. Hence we instrumented the script manually. The invariant is: `nice > 0` where nice is the priority. Note that on Linux, a priority greater than 0 is a low priority; most kernel processes run at priority 0. The script read the priority number from a configuration file, and stored it in a variable. Just before the shell code to lower a priority, the line was added that specified a nice value greater than zero. The trace file showed that after script execution, the nice value equaled ten, satisfying the invariant.

4.3.3. Property 14. [Partially Verified]

Property 14 requires that the agent listen for connections only in response to agent-initiated connections. To validate this, the code was instrumented at the places where the agent initiated communications with the server. There were four methods: "httpHead", "httpPost", "httpGetString", and "httpPut". The trace file output showed connection requests accepted in several places by the agent software. The initial entry was an "accept", which appears to violate the invariant. After discussion with the developers, it became clear that the agent opened a high-numbered port on startup, and listened for messages. When a message was received, the agent responded to the server. Thus, the trace file showed that the agent satisfied the invariant except for the invocation of the "Listener". The invariant needed to be rewritten to take this into account, but a limit of the TASpec langauge made this difficult to do in the time allotted (see below). The limit has since been removed.

However, the "Listener" port was discovered to be a potential avenue for a DoS attack when the traces for the beginning of network connections were reviewed. Network probes to the "Listener" port cause the agent to "wakeup" and check-in with the server. The "Listener" port provides the capability to check on the agent status to see if it is "alive" and cause it to check with the server for jobs to perform. Since the server can handle only a small number of simultaneous connections, agent connections are rejected when it reaches its connection limit. The agents will attempt reconnection if the server does not respond on check-in. Continual network probes to this port will create a DoS for agents trying to check for jobs, and for the server trying to respond to agent requests.

4.3.4. Properties 1-3. [Not Tested]

These properties require the use of SSL and the method "PlUtil::setSecurityProvidedIfRequired" be called whenever a URL beginning with "https://" is requested. If any initiation occurs without SSL being set, the invariant will fail. In fact, if one desires to

IEEE COMPUTER SOCIETY

require the url to begin with "https://", then the constraints of the TASPEC language require post-processing of the trace file to eliminate all initiate lines without "https://" at the beginning of the URL. The modification below eliminates the need for this post-processing.

4.3.5. Property 14. [Not Tested]

The testing for property 14 would ideally match the hosts referred to in the `initiate` predicates with the hosts from which connections were accepted. However, the TASPEC language did not support embedding Java code in the location specifications. Extraction of host names from the parameters could not be put into the predicates in Java code. Instead, one must put the full URLs into the `initiate` predicate, for post-processing to extract the host names. For the same reason, one cannot extract the host name from the socket name to put into the `accept` predicate. The modification described below eliminates the need for this post-processing.

4.3.6. Property 15-16. [Not Tested]

These properties deal with timeouts. In order for a timeout to be tested, one is looking for the property that the agent waits for an event, and the event either occurs or the timeout occurs. Thus, the properties would require that an event beginning and end be identified. For test purposes, the beginning of the event is indicated by the invocation of the method "xyzzy::event_begin()" and the end is indicated by the invocation of the method "xyzzy::event_end()". A timeout will cause an exception that invokes the method "xyzzy::timeout()".

5. Summary and PatchLink Response

The piloting of the MBV FMF and the PBT instruments on the PatchLink UNIX agent provided value to both the customer of the product and the vendor. The verification activities provided a higher level of assurance of the security of the agent for those security properties checked and tested, and subsequently verified.

The MBV FMF activity took a week of preparation to specify the properties in Linear Temporal Logic (LTL) and then program them into Promela language. Two days were spent on-site with the developers to clarify the specifications and to review potential violations of properties. The PBT activity took another week for preparation after providing properties and LTL. Also, two days were spent on-site with the developers to ensure that the properties were translated correctly into TASPEC.

5.1. PBT Issues

An issue uncovered with the PBT was that there needed to be refinements to the properties passed on to the PBT from the modeling activity. This issue was previously noted [7]. As a result of the Property 14 issue, a change to TASPEC allowed Java code to be put into the bodies of the specifications. Several other minor changes were made as a result of other inconveniences found during the PBT testing. Other issues uncovered showed that not all Java instantiations are the same on all operating system platforms. This issue impacted the testing activities. Enhancements to the PBT tool were made based on the pilot study to reduce these inconsistencies.

The PBT instrumenter was extended and updated as a result of the prototype activities. This eliminates the time-consuming work-arounds that had to be used during this testing. Between the familiarity gained from the source code, knowing what questions to ask, and the removal of some limits, it would be possible to test more invariants than was done during this test.

5.2. Verification Results

The results of the verification activity using the MBV FMF and the PBT, together and individually, indicate that the instruments may provide a higher level of confidence that the software artifacts meet specified properties, including security properties, when model-checked and tested.

The security properties that were modeled were properties 1-7, 14-16. The properties that were tested with the PBT were 5, 8, and 14. Properties 1-4, 6-7, and 15-16 were not tested for the reasons provided above. Properties 9-13 are dependent on the system kernel to ensure that the priorities of the services are correctly handled. Process 17 was observed but it is not possible to ensure that all events are logged as would require an *n* complete path for testing.

The verification results show that the security properties specified for the PatchLink UNIX agent that were modeled checked and tested were verified as holding and that the agent did not violate these properties. The PBT findings were due to imprecisions in the invariants or external factors. If these are eliminated, then the invariants will be satisfied and the properties will hold. While the verification does not prove that the agent is secure, it does provide a greater degree of confidence in its security if the environment is itself otherwise secure.

IEEE COMPUTER SOCIETY

5.3. PatchLink Response

.

PatchLink Corporation provided assistance to the verification team by assisting them as needed in their activities, and providing input and clarification to specifications and program code. As a result of the verification activity and the findings, PatchLink has implemented the following changes and measures:

1) PatchLink has always recommended that SSL be used on the PLUS web servers for communications. This can be configured by obtaining a valid, trusted website certificate from a certificate authority such as Verisign or Entrust or through the use of an internal PKI infrastructure.

2) The "Listener" port violates in part property 14. PatchLink has provided an option for this port to be turned off from the server or to have it off during installation of the UNIX agent. The agent now implements "safe defaults" by ensuring that this listen port functionality is disabled by default.

3) For property 18, a UNIX agent that runs as non-root has been released by PatchLink. JPL developed pre- and post-installation scripts that configure the agent to report into the server periodically and run at low priority.

5. Conclusion

The MBV and PBT verification of the UNIX user agent shows that the agent satisfies the properties model-checked and tested. In some cases the invariants in the PBT were violated, but these were due to imprecisions in the invariants or external factors. Once those causes were eliminated, the invariants were satisfied.

As with dynamic testing in general, the results of the PBT are valid only for the environment and test cases used in the experiments. In particular, no testing metrics or code coverage metrics were used to measure coverage. That said, Patchlink provided a quality verification environment that was similar to their own assurance facilities. It seems reasonable to assert that this environment is typical of their customers. The test cases involved an agent downloading and installing a patch in a manner that Patchlink believes is typical.

Both the MBV and PBT instruments were used together on the same properties. They were also used separately because some properties were more easily verified with only one of the instruments. This result bears out the earlier assessment that the instruments can be used in concert or separately for verification.

Both instruments require domain expertise to use. They were resource intensive to use at first. Once we built the model wrote the PBT properties, maintenance activities were more quickly addressed and required less time to complete.

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] H. Hussmann, Formal Foundations for Software Engineering Methods. Goos, G., Hartmanis, J., and van Leeuwen, J. (eds.), Lecture Notes in Computer Science, 1322. Berlin: Springer, 1997.

[2] S. M. Easterbrook and J. R. Callahan, Formal Methods for Verification and Validation of Partial Specifications: A Case Study. Report to NASA Independent Verification and Validation Facility, Fairmont, WV, 1996. Retrieved November 14, 2004, from www.cs.toronto.edu/~sme/papers/1998/NASA-IVV-97-010.pdf

[3] D.M. Nicol, W.H., Sandera, and S.T. Kishor, Model-Based Evaluation: from Dependability to Security, IEEE Transactions on Dependable and Secure Computing. Vol. 1, No. 1, 2004, pp. 48 – 65.

[4] PatchLink Corporation, www.patchlink.com.

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

[6] Holzmann, G. J., *The SPIN Model Checker: Primer and Reference Manual*. Boston, MA: Addison-Wesley.

[7] D.P. Gilliam, J.D. Powell, M. Bishop, Application of Lightweight Formal Methods to Software Security, Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. WETICE 2005, pp. 160-165.

[8] D.P. Gilliam, J.D. Powell, E. Haugh, M. Bishop, Addressing Software Security and Mitigations in the Life Cycle. Proceedings of the 28th Annual NASA Goddard IEEE Software Engineering Workshop (SEW), pp. 201 – 206.

[9] J. Rushby, Using Model Checking to Help Discover Mode Confusions and Other Automation Surprises, Reliability and System Safety. Vol. 75, No. 2, Feb. 2002, pp. 167-177.

[10] Klocwork, http://www.klocwork.com.

IEEE COMPUTER SOCIETY

[11] Secure Software, CodeAssure, http://www.securesoftware.com.

Proceedings of the 15th IEEE International Workshops on Enabling Technologies:Infrastructure for Collaborative Enterprises (WETICE'06)
0-7695-2623-3/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY