Integrating Contract-based Security Monitors in the Software Development Life Cycle

Alexander M. Hoole, Isabelle Simplot-Ryl, Issa Traoré

Integrating Contract-based Security Monitors in the Software Development Life Cycle*

Alexander M. Hoole[1], Isabelle Simplot-Ryl[2], Issa Traore[1]

[1]Dept. of Electrical and Computer Engineering
University of Victoria
P.O. Box 3055 STN CSC
Victoria, B.C. V8W 3P6
CANADA

[2]LIFL CNRS UMR 8022/INRIA Lille-Nord Europe
Universite de Lille I, Cite Scientifique
F-59655 Villeneuve d'Ascq Cedex
FRANCE

E-mail: alex.hoole@ece.uvic.ca isabelle.ryl@lifl.fr itraore@ece.uvic.ca

Abstract

Software systems, containing security vulnerabilities, continue to be created and released to consumers. We need to adopt improved software engineering practices to reduce the security vulnerabilities in modern systems. These practices should begin with stated security policies and end with systems which are quantitatively, not just qualitatively, more secure. Currently, contracts have been proposed for reliability and formal verification; yet, their use in security is limited. In this work, we propose a contract-based security assertion monitoring framework (CB_SAMF) that is intended to reduce the number of security vulnerabilities that are exploitable, spanning multiple software layers, to be used in an enhanced systems development life cycle (SDLC).

1. Introduction

Security has always been a hybrid of art and science as throughout history humans have attempted to protect valuable assets. Our modern information driven society has placed an increased value on data and the transfer and storage of information. More recently, in the last decade, industry and academia have pushed for more secure solutions for information technology assets and facilities as we have equally seen a rise in malicious hacking and security threats.

Many different approaches have been presented recently toward solving the problem of weak security; however, we obviously have not yet found a solution since security related attacks continue to persist.

Gary McGraw identifies three trends that have a large influence on the growth and evolution of the software security problem [13]. First, *connectivity* to the Internet has increased the number of attack vectors and the ease of which an attack can be made. Second, *extensibility* of software is allowing systems to grow in an incremental fashion which potentially adds new security vulnerabilities to existing systems. Lastly, the extensive increase of software *complexity* in modern information systems leads us to a greater number of vulnerabilities. These three trends will continue and lead us to one, hopefully obvious, conclusion. Security and dependability vulnerabilities must be resolved during design and testing before being released to the general public.

Recently, we have observed a promising shift in industry and academia to reduce security vulnerabilities during the software development life cycle (SDLC), rather than attempt to patch the problem after software is shipped [4, 8, 9, 13]. If we can reduce *security defects* early in the SDLC we reduce not only the number of vulnerabilities but also the risk of attack.

While there are areas being researched which target specific areas of security during the systems development life cycle (SDLC), a methodology for testing security across multiple software layers is still lacking. We propose a contract-based security assertion monitoring framework (CB_SAMF) that is intended to reduce the number of security vulnerabilities that are exploitable, spanning multiple software layers, to be used in an enhanced SDLC.

*This work is partially supported by CPER Nord-Pas-de-Calais/FEDER Campus Intelligence Ambiante.

The following section will review SDLC and how it relates to security, Section 3 discusses modeling techniques related to security, Section 4 introduces our proposed approach, Section 5 discusses our contract model, Section 6 expands on the benefits of contracts for security, and Section 7 provides our concluding remarks.

2 SDLC and Security

Security policy documents are often used by organizations to specify the laws, rules, practices, and principles that govern how to manage, protect, and transfer sensitive information. These policy documents represent a corner stone from which software requirements can be built. Requirements in turn drive most modern software/system development life cycles. During the SDLC there are many opportunities to reduce security vulnerabilities.

A SDLC is typically an iterative and recursive process which clearly identifies the stages that should lead a successful software project through its entire development life cycle. We are interested with integrating security into every phase of the SDLC. In fact, several tools and methodologies have already begun to integrate themselves accordingly. We believe, however, that there is a great deal of work remaining in this area.

The SDLC is still lacking models, methods, and tools that assist in creating more secure and reliable software products. The audience for this work includes individuals and teams fulfilling the following roles during a SDLC: analyst, architect, developer, tester, maintainer, user, and support. Essentially, all of the development-related stake holders in the SDLC.

Recently Serpanos and Henkel asserted that a unified approach to dependability and security assessment will let architects and designers deal with issues in embedded computing platforms [18]. The observation that security and dependability are interrelated is an important one. Serpanos and Henkel differentiate the two based on security flaws being problems that are exploited on purpose, while flaws which are exploited by accident would be qualified as dependability problems. It would be interesting to have a framework that can support both dependability and security. Thus, we have kept dependability in mind while designing our framework; however, we focus on security vulnerability monitoring since it is our primary concern.

The goal of our research is to create new methods, models, and tools that integrate into the phases of the SDLC to create more secure software. We cannot always depend on the consumer to have sufficient protection mechanisms in place on their systems.[1] We need to take a more active role during development to ensure software ships fewer security vulnerabilities.

A modified form of the SDLC is depicted in Figure 1 showing how various security activities can be integrated into the iterative and recursive SDLC. Existing SDLC hybrids integrate some of the steps identified in Figure 1 such as those put forward by CERT, Microsoft's Michael Howard and Steve Lipner [9], and others. Nothing has been identified to date that guarantees security in software systems; however, our aim is to help reduce the risk associated with security vulnerabilities.

3 Modeling

For many years software developers have been using methodologies meant to simplify and standardize the SDLC. One notation that has met with a great deal of success, in several methodologies, is the Unified Modeling Language(UML). UML does not handle all analysis, design, and implementation requirements for all projects. For example, UML is a natural fit for most object oriented languages; however, not all projects demand an object oriented approach. Projects that require high performance, and a low memory footprint, are typically implemented in non-object oriented languages such as C.

Several diagrams that are used in UML are useful in the broader spectrum of all software design projects. For instance, use case diagrams are very useful in identifying the main functions of a software artifact. In fact, use cases provide the earliest opportunity to identify security risk in a new SDLC for a given application (other than general risk analysis).

Recently, a more modern addition to use cases, called misuse cases, has been created. Misuse cases, also known as abuse cases, can be used during requirements analysis [1, 7, 17] leading to a more complete understanding of potential security risks that need to be mitigated. Hope, McGraw, and Antón also mention that misuse cases can be over-used and can lead to identification of a fairly large set of misuse cases that may have little impact on security [7]. With the knowledge of subject matter experts and security analysts these misuse cases need to be prioritized to balance risk and cost. During development many risks can be completely mitigated based on the early warnings of the misuse cases. We must recognize, however, that the probability of a particular misuse may not be completely understood and that some risks may not be identified until later in the SDLC. It would be useful to have a mechanism that can identify these security threats in the code and allow for a monitoring system to be implemented to capture and trace any possible misuse.[2]

[1] Consumers often employ intrusion detection systems, firewalls, and other products to help reduce security risks.

[2] An example of such an approach would be to use the output of static analysis security tools as the basis of misuse case creation.

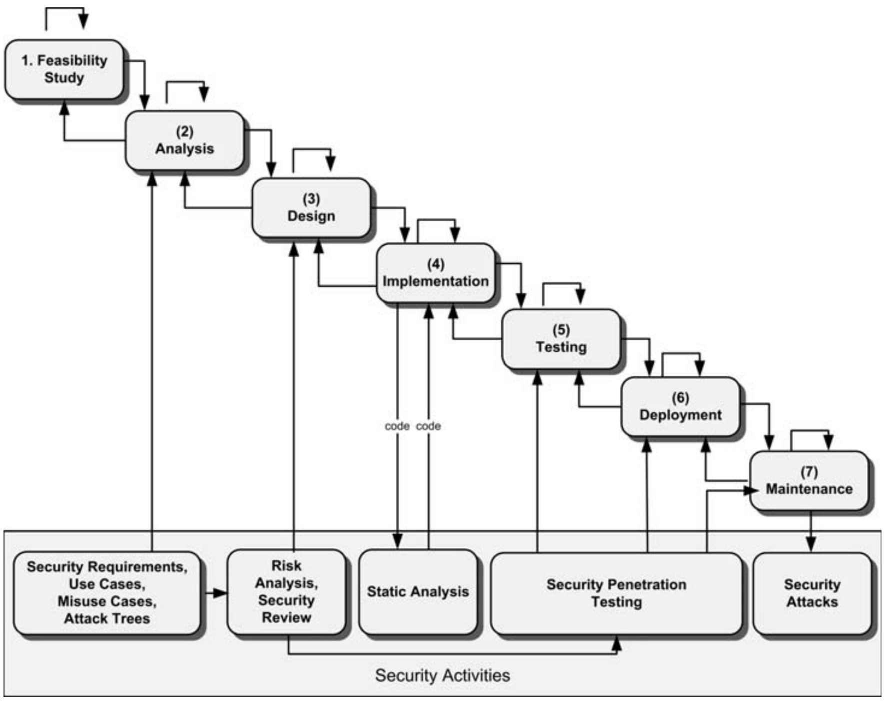

Figure 1. Security activities integrated into the typical waterfall SDLC. Regular SDLC steps are numbered and linked in diagonal. Security activities are shown horizontally.

Misuse case diagrams can be used to expose a wide variety of threats including privacy violation, denial of service, privilege escalation, identity or information theft, and network based attacks. As with use case diagrams, misuse case diagrams are continually reviewed and revised throughout the SDLC. The components that make up a misuse case are documented already in [1, 7, 17].

Once misuse cases have been identified we can then proceed with the identification of security violation scenarios. One technique for identification of these violation scenarios is the use of an attack tree. Each depth first traversal of an attack tree will identify possible violation scenarios [15].

4 Proposed Approach

Now that we have discussed some of the methods for identifying potential vulnerabilities, we propose a model for monitoring applications for security violations during the middle phases of the SDLC which also allows for the collection of forensic data based on the prioritized security risks identified earlier in the SDLC.

This monitoring framework can be integrated early during SDLC. In Figure 2, we depict how the security policy document is used as part of the processes identifying the security requirements. Security requirements are then used during the identification of misuse cases (along with normal use cases) that are intended to identify potential vulnerabilities. Once prioritized, these misuse cases can then drive the creation of attack trees which further identify intrusion scenarios. The intrusion scenarios can then be used during design and testing to create sequence diagrams and associated test cases. Finally, during implementation, sequence diagrams can be generated which identify security vulnerabilities (for example, system/function calls that have known vulnerabilities). Once a vulnerability has been identified, a ”contract” can be created using assertions and additional rules to guard against, or verify, a given vulnerability. These contracts can then in turn be used to generate security probes that are used during execution to track forensic data in our monitoring framework (CB_SAMF).

Consideration should be given as to whether or not output formats from existing tools, such as static analysis tools, may be translated into a format that may be used by the assertion monitoring framework.

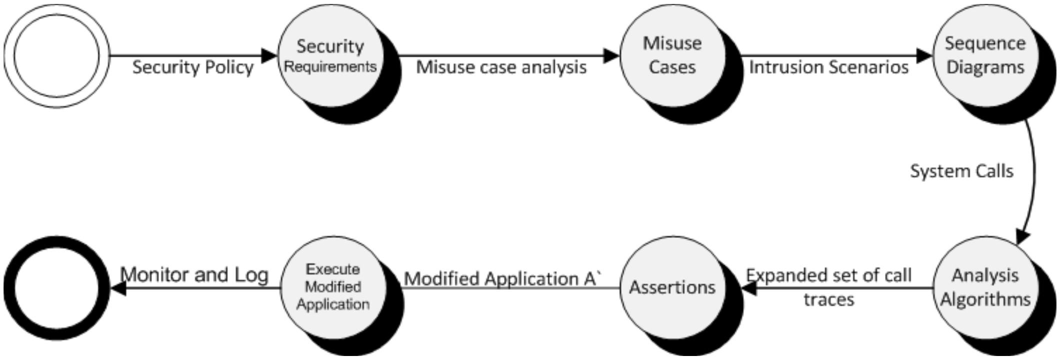

Figure 2. System flow diagram leading to the use of contracts and monitoring probes.

Ultimately the focus of the initial work will be on the last three nodes of Figure 2 by creating and consuming a contract, generating the assertion probes, monitoring assertions, and reacting appropriately using the monitoring framework.

5 Contract Model

The notion of a contract used in software engineering is not a new idea [6, 10, 11, 14]. When used for security, however, we must look outside of the basic preconditions and postconditions that are often used when implementing systems using contracts and look carefully at what properties need to be specified in a contract to improve security. Historically, the precondition specifies when it is appropriate to call a particular feature (function/method), while a postcondition specifies what is true after a particular feature is called (what has been accomplished by the function/method). [3]

Our definition of contract needs to bind the caller and callee to deal with additional properties involving timing, property values, and other events.[4] For example, a contract that is specified for a supplier *X* is consumed by a consumer *Y* guarantees that *X* has fulfilled the postcondition(s), provided that *Y* has satisfied the precondition(s). Thus, the contract provides protection for both parties. The consumer is protected from the supplier since the postconditions have been guaranteed by the supplier. The supplier is protected from the consumer since the preconditions have been guaranteed by the consumer.

Contracts, as proposed by Meyer, are not suitable for security monitoring.[5] The require, guarantee, and references fields of the contract, that correspond to the pre, post, and invariants, do not handle all of the necessary attributes of security defects. In particular, we would propose the addition of several new contractual fields including *context*, *history*, and *response*. Context is required since the basic reliability contract above does not factor environmental influence. History is required since security vulnerabilities are often complex and are sometimes the result of a series of actions which may occur in parallel. Both context and history can be useful when dealing with DoS and race-condition vulnerabilities. Finally, response is required so that we can choose how a particular assertion is handled when an exploitation is detected. We desire the ability to deal with security assertion failures, not just detect them as would be the case if we used the form of contract proposed by Meyer.[6]

Our form of contract includes the following fields:

- Requirements - in the form of preconditions (PRE)
- Guarantees - in the form of postconditions (POST)
- References - in the form of invariants (INV)
- Context - in the form of relevant environmental information (CONT)
- History - in the form of some knowledge keeping construct (HIST)

[3] Many pre and postconditions are more to do with robustness than security.

[4] The definition of **binding contract**: The legal agreement between two or more entities to perform and/or not perform a set of actions.

[5] For example,under normal contracts, a false precondition does not guarantee that the system will not process the input. It may still allow certain types of attacks such as buffer overflows to continue.

[6] The concept of resumption and organized panic for exception handling, used by Meyer, could also fall under our broader response category [14].

- Response - in the form of a reactive measure (RESP)

Work done by Barringer *et al* on program monitoring and rule-based runtime verification has exposed interesting results [2, 3]. Specifically, the work on linear temporal logic (LTL) and program states has been core to several attempts towards runtime verification and is a promising candidate for the notation of our contracts.

Each contract (*C*) will contain a breakpoint (*B*) and one or more assertions (*A*). A breakpoint identifies a monitoring location or symbol in the target application. For example, a contract should be able to specify a target function in a program which affects the state of an assertion. The assertion is a rule which must remain true at the breakpoint. Each assertion has associated with it zero or more of the security contract extensions (*E*) mentioned above (context, history, and response). An assertion can take on one of the following three forms: precondition (*PRE*), postcondition (*POST*), or invariant (*INV*). We do not represent the assertions types separately since they all take the same form. Each assertion is composed of zero or more rules (*R*), relating to the target (remember the breakpoint *B*), and zero or more monitors (*M*). The rules, monitors, and extensions are individually named (*N*). A rule specifies a property of the state of the program which needs to remain true, while a monitor enforces one or more rules. The quantifiers *min* and *max* represent liveness and safety properties respectively and are important for the boundary cases of a monitor trace. The body of every rule and monitor is specified as a boolean valued formula of the syntactic category *Form*.[7] Therefore, each contract may be instantiated using the following grammar[8]:

$$C := B\ (A\{E\})\ \{A\{E\}\};$$
$$E := \{\mathit{CONT}\} \mid \{\mathit{HIST}\} \mid \{\mathit{RESP}\};$$
$$A := \{R\}\{M\};$$
$$R := \{\underline{\text{max}}|\underline{\text{min}}\}\ N(T_1 x_1, ..., T_n x_n) = F;$$
$$M := \underline{\text{mon}}\ N = F;$$
$$T := \underline{\text{Form}} \mid \textit{primitive type};$$
$$B := \textit{symbol} \mid \textit{HEX address};$$
$$F := exp|\underline{\text{true}}|\underline{\text{false}}|\neg F|F_1 \wedge F_2|F_1 \vee F_2|F_1 \rightarrow F_2| \odot F| \ominus F|$$
$$F_1 \cdot F_2|N(F_1, \ldots, F_n)|x_i;$$
$$CONT := \underline{\textit{env}}\ N \mid \underline{\textit{res}}\ N;$$
$$HIST := \underline{\textit{trace}}\ N \mid \underline{\textit{runningsum}}\ N \mid \underline{\textit{runningavg}}\ N;$$
$$RESP := \underline{\textit{core}}\ N \mid \underline{\textit{term}}\ N \mid \underline{\textit{kill}}\ N \mid \underline{\textit{log}}\ N;$$

When defining rules, the max prefix indicates that a given rule defines a safety property and min indicates that a rule is a liveness property [3, 16].[9] We have also tentatively defined possible extended behaviors for context, history and response elements and may extend these in the future. Context may specify environmental or resource information (external to the program) which is needed by the contract. History may contain trace data or statistically relevant information for the contract. Finally, response may specify an action to perform an assertion is violated.[10]

From this definition it is possible to use multiple separate monitors or redirect multiple rules to the same monitor.

6 Benefits of Contract for Security Monitoring

Targeting the identification, verification and removal of security vulnerabilities from systems is not a trivial task. We chose the notion of contracts for an assertion framework so that we can state precise properties about a system without having to

[7] This notation is derived from linear temporal logic (LTL) and is inspired by the EAGLE framework that was proposed by Barringer *et al* [2, 3].

[8] Each line is a Extended Backus-Naur Form (EBNF) production. Following is a simplified description of EBNF notation that we have used:
:= meaning ”is defined as”
| meaning ”or”
, meaning concatenation (used to separate items in a sequence)
{ } meaning zero or more times
{ }- meaning one or more times
[ ] meaning optional item
( ) meaning grouping
; marks the end of a rule

[9] Safety properties state that if a behavior is unacceptable any extension of that behavior is also unacceptable. Liveness properties state that for a given requirement, and any finite duration, the behavior can always be extended such that it satisfies the requirement[16, 12].

[10] Possible responses include the following: core=produce a core dump, term=terminate the task, kill=kill the task, log=produce an audit report for the event.

modify the code directly. In order to understand the benefits of these contracts for security monitoring we will briefly discuss a variety of common security vulnerabilities. An example set of common security problems found in systems is as follows:

- Exploitable Logic Error
- Inadequate Concurrency Control
- Weak Dependencies/Altered Files
- Inadequate Parameter Validation Incomplete/Inconsistent
- Inadequate Authentication/Authorization/Identification
- Implicit Sharing of Data and Data Leakage

As we progress with this work we expect that a wide variety of vulnerabilities should be covered by contracts. Exploitable logic errors are difficult to track down; however, if we can identify environmental, historical, or timing information related to the expected behavior, contracts can be written to detect misuse. Parameter validation issues can be handled by our pre and post conditions. Concurrency, accountability, and protocol issues can be tracked through the use of historical, environmental, pre and post conditions. Finally, the addition of historical and environmental assertions should allow us to track vulnerabilities related to weak dependencies and data leakage. Furthermore, to give an idea of the types of attacks we should attempt to counter, a listing of network related attack classes is as follows (derived from [5]):

- Password Stealing
- Bugs and Back Doors
- Protocol Failures
- Exponential Attacks Viruses and Worms
- Botnets
- Social Engineering
- Authentication Failures
- Information Leakage
- Denial-of-Service Attacks
- Active Attacks

Contracts are not suitable for dealing with all types of attacks. For example, password stealing can occur through the use of a network sniffer or through the use of social engineering techniques. The ability of an attacker to passively monitory network traffic will not be prevented through the use of contracts; however, we can use contracts to ensure that security properties of our systems (derived from our initial security policies) are observed. In the case of password stealing, the password should never enter a public network in clear text and the protocol used for authentication should not be subject to replay attacks. These are properties for which we can design contracts.

7 Conclusion

Our enhanced version of contracts provides a novel way to propagate requirements-based security assertions through the SDLC. Some techniques, such as misuse cases, attack trees, and static analysis, are already providing ways of identifying potential vulnerabilities during the early phases of the SDLC; however, these approaches can lead to a high rate of false-positives which consume resources. Our (CB_SAMF) is able to help reduce vulnerabilities in multi-layered systems by not only providing a way to detect if a particular contract is violated, but also provides reactive measures.

References

[1] I. Alexander. Misuse cases: Use cases with hostile intent. *IEEE Software*, 20(1):58–66, 2003.

[2] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. *ipdps*, 17:264b, 2004.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification, 2004.

[4] M. Bishop. *Computer Security: Art and Scienc*. Addison-Wesley, Boston, MA, USA, 2003.

[5] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[6] A. Février, E. Najm, and J.-B. Stefani. Contracts for odp. In *ARTS '97: Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software*, pages 216–232, London, UK, 1997. Springer-Verlag.

[7] P. Hope, G. McGraw, and A. I. Anton. Misuse and abuse cases: Getting past the positive. *IEEE Security and Privacy*, 02(3):90–92, 2004.

[8] M. Howard. Building more secure software with improved development processes. *IEEE Security and Privacy*, 2(6):63–65, 2004.

[9] M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.

[10] J. C. M. Jr. Programming by contract. *Computer*, 29(3):109–111, 1996.

[11] L. Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32(1):32–45, 1989.

[12] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[13] G. McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[14] B. Meyer. Applying ”design by contract”. *Computer*, 25(10):40–51, 1992.

[15] A. Moore, R. Ellison, and R. Linger. Attack modeling for information security and survivability, 2001.

[16] D. K. Peters and D. L. Parnas. Requirements-based monitors for real-time systems. *SIGSOFT Softw. Eng. Notes*, 25(5):77–85, 2000.

[17] G. Peterson and J. Steven. Defining misuse within the development process. *IEEE Security and Privacy*, 04(6):81–84, 2006.

[18] D. Serpanos and J. Henkel. Dependability and security will change embedded computing. *Computer*, 41(1):103–105, 2008.