Catalog of Metrics for Assessing Security Risks of Software throughout the Software Development Life Cycle

Khalid Sultan, Abdeslam En-Nouaary, Abdelwahab Hamou-Lhadj
Department of Electrical and Computer Engineering
Concordia University, Montreal, Canada
{k_sultan, ennouaar, abdelw}@ece.concordia.ca

Abstract

In this paper, we present a new set of metrics for building secure software systems. The proposed metrics aim to address security risks throughout the entire Software Development Life Cycle (SDLC). The importance of this work comes from the fact that assessing security risks at early stages of the development life cycle can help implement efficient solutions before the software is delivered to the customer. The proposed metrics are defined using the Goal/Question/Metric method. It is anticipated that software engineers will use these metrics in combination with other techniques to detect security risks and prevent these risks from becoming reality. This work is part of a larger research project that aims at examining the concept of "Design for Security". The objective is to investigate software engineering techniques to support security requirements from the very beginning of the development process.

Keywords: Software security, security metrics, software development lifecycle, design for security

1. Introduction

Nowadays, security problems involving computers and software are frequent, widespread, and serious. Therefore, the production of secure software is a must in the interconnected electronic world of today [1, 2, 3, 4, 5, 6]. Developing secure software requires that the developers address security issues at all phases of the software development process. To cope with the situation, security metrics could help. They are powerful techniques that can help software designers and developers integrate security features into their systems from the very beginning in the development lifecycle [7, 8, 9, 10, 11].

The use of metrics has recently received a lot of attention. They provide useful data that can be analyzed and utilized in technical, operational, and business decisions across an organization. Metrics, in general, assist developers in meeting overall mission goals such as continuity of operations, safety, reliability, and security [6, 10, 12, 13].

In this paper, we have put forward a set of new metrics for building secure software. This set will update software engineers throughout the entire development process about the security level of their software so that they get the necessary feedback earlier before releasing their software. The reason of developing these metrics is due to the fact that the idea of introducing such metrics for each phase of the (SDLC) has never been materialized in the past. Therefore, we believe that our contribution will be a step forward in achieving the goal of producing secure software. The proposed suite of metrics is further divided into subgroups where each subgroup corresponds to a particular phase of the software development lifecycle.

The rest of this paper is organized as follows: In section 2, we discuss the approach that has been applied while gathering the metrics, as well as the catalog of metrics for assessing security risks of software throughout the software life cycle. Finally, we conclude the paper and present future directions.

2. Catalog of Metrics

In this work, the Goal/Question/Metric (GQM) approach has been applied to identify the appropriate measurable goals for achieving software security, and theoretically valid measurements for indicating the achievement of security goals [14]. GQM is one of the best-known and widely used mechanisms for defining measurable goals, and the appropriate measurements, which would characterize the achievement of those goals. The goal of the proposed metrics can be stated as follows:

Enable software engineers to better assess and detect security risks during the entire life cycle of the software system

We have designed questions that aim at achieving the above goal. The metrics are grouped according to each phase in the SDLC. They are presented along with the questions they address in the following subsections.

2.1. Requirement Phase

This category of metrics aims at answering the following question:

Q1: How is security assessed during the requirements phase?

978-0-7695-3126-7/08 $25.00 © 2008 IEEE
DOI 10.1109/ISA.2008.104

Knowing this will allow software engineers to assess whether security requirements have been properly considered during the analysis phase. The following metrics make this more precise.

Total number of security requirement [Nsr]: This metric measures the number of security requirements identified during the analysis phase of the application. We will use this metric as a baseline for defining the remaining metrics.

Ratio of security requirements [Rsr]: This metric measures the ratio of the number of requirements that have direct impact on security to the total number of requirements of the system. Knowing this will allow software engineers to assess the level of security supported by the system. A good example of a security requirement is when an organization requests that an authentication mechanism be supported and that the identity of the users of the system must be verified before usage.

More formally, `Rsr` is defined as follows:

- `R`: A set of all requirements of the system
- `SR`: A set of security requirements of the system. `SR` is a subset of `R`.

```
Rsr = |SR| / |R|
```

Number of omitted security requirements [Nosr]: This metric measures the number of requirements that should have been considered when building the application. Identifying the missing security requirements can be accomplished using different techniques. For example, it can be achieved by comparing the stated security requirements of the application to the ones imposed by a specific security standard such as the Common Criteria for Information Technology Security Evaluation (ISO/IEC 15408), which is an international standard for identifying and defining security requirements. Another technique for detecting missing security requirements is by using benchmarking techniques – Comparing the actual application to the other applications that require similar degree of security. Knowing the number of security requirements that have not been considered will allow software engineers to assess the risks of possible attacks, and understand what would be the impact of these attacks on the system.

Ratio of the number of omitted security requirements [Rosr]: This metric measures the ratio of the number of security requirements that have not been considered during the analysis phase (i.e. Nosr) to the total number of security requirements identified during the analysis phase (Nsr); the higher the ratio, the more vulnerable the system.

More formally, `Rosr` is defined as follows:

```
Rosr = Nosr / (Nosr + Nsr)
```

`Rosr` varies from 0 to 1. If it reaches 0 then the security of the system is well established. The metric converges to 1 if a large number of security requirements have not been considered. This is a good indicator of the existence of many weaknesses within the system.

2.2. Design Phase

This category of metrics aims at answering the following question:

Q2: How is security assessed during the design phase?

The objective of the metrics belonging to this category is to assess whether security has been properly dealt with during the design phase. To achieve this, we propose the following metrics.

Number of design decisions related to security [Ndd]: During the design phase, it is common to end up with multiple solutions to the same problem. Software engineers make many design decisions in order to choose among alternative design solutions. The proposed metric aims at assessing the number of design decisions that address the security requirements of the system. This metric can help software engineers assess the importance given to security during the design phase.

Ratio of design decisions [Rdd]: This metric measures the ratio of the number of design decisions related to security to the total number of design decisions of the entire system. The objective is to assess the portion of design dedicated to security. `Rdd` can be computed as follows:

```
Rdd = Ndd / Nd
```

`Nd` is the number of design decisions applicable to the entire system.

Number of security algorithms [Nsa]: This metric measures the number of security algorithms that are supported by the application. Examples of security algorithms include algorithms found in the area of cryptography [3, 4], authentication techniques [3, 4], etc.

Number of design flaws related to security [Nsdf]: Security-related design flaws occur when software is planned and specified without proper consideration of security requirements and principles. For instance, clear-text passwords are considered as design flaws. Design flaws can be detected using design inspection techniques (e.g., design reviews). Identifying the number of design flaws related to security can help detect security issues earlier in the design process.

Ratio of design flaws related to security [Rdf]: This metric computes the ratio of design flaws related to security to the total number of design flaws applicable to the whole system.

More formally, Rdf can be computed as follows:

$$Rdf = Nsdf / Ndf$$

Ndf is defined as the number of design flaws applicable to the entire system.

2.3. Implementation Phase

This category of metrics aim at answering the following question:

Q3: How is security assessed during the implementation phase?

Number of implementation errors found in the system [Nerr]: This metric measures the number of implementation errors of the system. It is important to note that the vulnerability of the system is often related to implementation errors. This metric will be used as a baseline metric for defining the remaining metrics.

Number of implementation errors related to security [Nserr]: This metric measures the number of implementation errors of the system that have a direct impact on security. One of the most common security related implementation errors is the buffer overflow [15, 16].

Ratio of implementation errors that have an impact on security [Rserr]: This metric measures the ratio of the number of errors related to security to the total number of errors in the implementation of the system (i.e. Nerr).

More formally, Rserr is defined as follows:

$$Rserr = Nserr / Nerr$$

Rserr converging to 1 indicates the presences of a significant number of implementation errors that would compromise the security of the system.

Number of exceptions that have been implemented to handle execution failures related to security [Nex]: The objective of this metric is to measure the number of exceptions that have been included in the code to handle possible failures of the system due to an error in a code segment that has an impact on security. Error handling mechanisms can be used to indicate how the system should react in the presence of an error in a code segment related to security. The absence of such handlers renders the system less robust to such errors.

Number of omitted exceptions for handling execution failures related to security [Noex]: This metric computes the number of missing exceptions that have been omitted by software engineers when implementing the system. These exceptions can easily be determined through testing techniques.

Ratio of the number of omitted exceptions [Roex]: This metric measures the ratio of the number of omitted exceptions (i.e. Noex) to the total number of exceptions that are related to security. More formally:

$$Roex = Noex / (Noex + Nex)$$

Knowing this will help software engineers assess the quality of the code in responding to failures of code segments that implement security algorithms.

2.4. Testing Phase

This category of metrics aims at answering the below question. We also present the metrics that address this question.

Q4: How is the software system tested for security?

Ratio of security test cases [Rtc]: This metric measures the number of test cases designed to detect security issues to the total number of test cases of the entire system. Knowing this will help software engineers determine the amount of testing for security that the system has undergone.

More formally, we define Rtc as follows:

- T: A set of all test cases of the system
- TS: A set of test cases that address security issues

$$Rtc = |TS| / |T|$$

Ratio of security test cases that fail [Rtcp]: This metric determines the number of test cases that detect implementation errors (i.e. the ones that fail) to the total number of test cases, designed specifically to target security issues. This metric should be combined with the implementation related metrics to assess the robustness of the system to counter malicious attacks.

More formally, we define Rtcp as follows:

- TP: A set of the security related test cases that pass
- TF: A set of the security related test cases that fail

$$Rtcp = |TF| / (|TP| + |TF|)$$

2.5. Maintenance Phase

This category of metrics aims at answering the following question:

Q4: How is security considered during software maintenance and evolution?

Ratio of software changes due to security considerations [Rsc]: This metric measures the number of changes in the system triggered by a new set of security requirements. Software changes due to security considerations include patches that are released after a system is delivered, or any other security updates. This metric helps identify the amount of work performed in order to keep the application secure.

Comparing the changes made with respect to security to the entire number of changes performed on the system is a good indicator of the importance given to security during software evolution.

More formally, we define Rsc as follows:

- Nc: The number of changes of the entire system
- Nsc: The number of changes triggered by new security requirements

$$Rsc = Nsc / Nc$$

Ratio of patches issued to address security vulnerabilities [Rp]: Not all software changes are related to security. This metric measures the ratio of the number of patches that are issued to address security vulnerabilities to the total number of patches of the system. Knowing this will allow software engineers to assess the effectiveness of the analysis, design, and implementation techniques used to address security. If the number of security related patches is high then the system was built with a poor consideration to security issues. More formally, we define Rp as follows:

- Np: The number of patches of the entire system
- Nsp: The number of patches related to security

$$Rp = Nsp / Np$$

Number of security incidents reported [Nsr]: This metric measures the number of incidents related to security that are reported by the users of the system. Reporting the incidents helps outline the actions to be implemented in response to these incidents. In addition, it has been shown that the majority of security incidents results from defects in software requirements, design, or code [5]. Similar to the previous metric, a high value of Nsr might require from software engineers to revisit the way security has been handled throughout the various phases of the software process.

3. Conclusion and Future Work

In this paper, we proposed a catalog of metrics that can be used to assess security risks at different stages of the software development process. The objective is to allow software engineers to have a good understanding of the weaknesses and exploits that a software system might have, and design the proper solutions before shipping it to customers.

The proposed metrics should be considered as a framework that can be used as they are presented in this paper, or from which other metrics can derive. In addition, the proposed metrics are inter-connected. For example, knowing that the number of security related test cases that fail is a good indicator of the existence of many errors in the system. The metrics defined in the implementation phase will help software engineers dig deeper and further assess the quality of the implementation to support security.

As future work, there is need to apply these metrics to various software projects. The outcome will not only reveal the vulnerability of the systems but also the strengths and weaknesses of the software engineering techniques to support security throughout the software process.

An important area of future work would be to evaluate the usefulness of these metrics to software engineers. We also need to investigate how these metrics can be supported by CASE tools.

4. References

[1]. G. McGraw, Software Security, *Journal of IEEE Security and Privacy*, 2004, 80-83.

[2]. D. P. Gilliam, T. L.Wolfe, J. S. Sherif, Software Security Checklist for the Software Life Cycle, *In Proc. of the 12th International Workshops on Enabling Technologies*, 2003, 243-248.

[3]. J. Viega, G. McGraw, Building Secure Software: How to Avoid Security Problems the Right Way (Addison-Wesley, 2002).

[4]. N. Davis, "Developing Secure Software", *Software Tech News: Secure Software Engineering, 8(2)*, 2005.

[5]. M. Bishop, Computer Security: Art and Science (Addison-Wesley, 2002).

[6]. G. McGraw, Software Security: Building Security In (Addison-Wesley, 2006).

[7]. J. A. Chaula, L. Yngström, and S. Kowalski, Security Metrics and Evaluation of Information Systems Security, *In Proc. of the 4th Annual Conference on Information Security for South Africa*, 2004.

[8]. M. Swanson, N. Bartol, J. Sabato, J. Hash, and L. Graffo, Security Metrics Guide for Information Technology Systems, *NIST Special Publication 800-55, National Institute of Standards and Technology,* 2003.

[9]. J. C. Munson, Software Engineering Measurement (Auerbach Publications, 2003).

[10]. S.C. Payne, A Guide to Security Metrics, *SANS Security Essentials GSEC Practical Assignment*, 2001.

[11]. R. Scandariato, B. D. Win, and W. Joosen, Towards a Measuring Framework for Security Properties of Software, *In Proc. of the 2nd workshop on Quality of Protection*, 2006, 27-30.

[12]. P. Goodman, Software Metrics: Best Practices for Successful IT Management (Rothstein Associates Inc., 2004).

[13]. S. A. Whitmine, Object Oriented Design Measurement (Wiley Computer Publications, 1997).

[14]. V. R. Basili, G. Caldiera and H. D. Rombach, Goal Question Metric Paradigm, *In J. J. Marciniak (ed.), Encyclopedia of Software Engineering 1*, New York: John Wiley & Sons, 1994, 528-532.

[15]. E. Haugh and M. Bishop, Testing C Programs for Buffer Overflow Vulnerabilities, *In Proc. of the Network and Distributed System Security Symposium,* 2003, 123–130.

[16]. C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*, In Proc. of the DARPA Information Survivability Conference and Exposition*, 1999, 119 - 129.