A Method Level Security Metrics Suite for Java Programs

Sree Ram Kumar T

Research Scholar,
Madurai Kamaraj University,
Madurai

Dr K Alagarsamy

Associate Professor,
Computer Center, Madurai Kamaraj University, Madurai.

Abstract

One of the biggest challenges faced by software engineers today is the engineering of secure software. Attempts are being made to apply the principles originally proposed for the engineering of “quality” software to security. One of such principles is related to the development and usage of “metrics” which are measures serving as indicators of how much of “something” software possesses. Security metrics attempt to measure the “amount” of security a software has. In this paper, we propose some metrics, which apply at the source code level that can serve as a guide for software developers in identifying the most vulnerable parts of the source code. We also demonstrate the validity of the proposed metrics through empirical results.

Keywords: Security Metrics, Security Source Code Metrics

1. Introduction

In the early decades of the software industry, the engineering of “quality” software was a big challenge to the software developers and managers of the period. Many sound principles that aid in the development of quality software were proposed and the adoption of these principles lead to a perceivable improvement in the quality of the software being engineered. Such principles related both to the product and the process employed to engineer the product. Examples of such principles include reviews, inspections, and Software Quality Assurance.

While addressing this issue of engineering of quality software, developers and managers had a desperate need to know how much of quality is present in the software they are attempting to engineer so that the can tune their products and processes to ensure that they result in the required quality. The most easily graspable indicators were measures called metrics. For instance, if the developer has some means to measure the reliability of the program he is developing and sees that his program has 70% reliability, he can reassess his program and find some means to improve the reliability.

Some researchers believe that any attempt to quantify the quality of a software product or process is futile either because it is difficult or because it will lead to wrong conclusions. But many researchers agree that if the metrics are computed correctly in a given context, they will indeed be indicators of the attribute they attempt to measure. Voas et al. [1] state that many metrics fail not because they are incorrect but because their value is often over sold. As long as the meaning of metrics is taken in a relative context, there is nothing inherently wrong in using them.

2. Secure Software Engineering

With the penetration of software into many areas such as business, commerce and governance, security began to evolve as one of the major challenges confronting the software engineers. The development of distributed computing in general, and the World Wide Web in particular compounded the problem greatly, as software was no longer restricted to a single machine. Many early attempts to consider security as just another quality attribute expected of software[2], failed because such an approach was clearly insufficient to address the big threat posed by software security compromises.

There have been attempts to incorporate many principles originally proposed for ensuring software quality to the engineering of secure software. [7] is an attempt to draw on the
parallels between the challenges posed by Software Reliability and Software Security. Many software engineering principles were extended to accommodate the new challenges presented by the security aspect. For Example, Software Quality Assurance is now complemented by Software Security Assurance. The Designer is now required to assess his design not only for its conformance to the explicitly stated functional requirements but also for security.

3. Security Metrics

There has been a great deal of interest and research in the area of measuring the security of a software. A comprehensive set of security metrics that can be computed objectively has always remained elusive for the researchers. But such metrics, it is hoped, will be of great value to developers and managers in their endeavor to engineer secure software. Just as some researchers have a different opinion the metrics applying to the quality of software, many are also opposed to attempts of development and usage of security metrics either because they are baffled by the inherent complexity and difficulty of this task[3] or because they are pessimistic about the correctness and accuracy of the developed metrics. But we believe that, as long as such security metrics are interpreted as relative and not absolute indicators of software security, they can indeed be of much help to the software community.


Security Metrics can be computed at any stage of Software Engineering. For example, some metrics can be computed during the analysis phase as the number of ambiguities or inconsistencies discovered pertaining to security requirements. Some can be computed at the design phase. Generally, metrics that can be computed at the earlier phases are considered to have more utility as if the computed values of such metrics indicate anything wrong, corrections can be applied immediately and it is generally easy to make corrections at early stages.

Source Code is one of the most attractive candidates for measurement as it is the direct embodiment of the software. Many metrics that attempt to measure the quality of the software using the source code have been proposed and studied [4,5]. Many static analysis tools that automatically scan the source code and compute the metrics have also been developed [6]. These metrics have the only disadvantage that they can be computed only after the programs have been developed which is a late part in the software engineering process.

In attempts to develop security metrics, those relating to the source code seem to be the most easily obtainable and objective. In this context, Voas et al propose a fault injection based security metric[1] and we have in an earlier work, suggested that this approach can be enhanced to give more accurate estimates using Mutation Testing.

5. Proposed Metrics

In this section, we propose some metrics that can be computed from the source code. The computation of some of these metrics can be automated while for the others, a manual static analysis of the source code is essential. All the proposed metrics are computed at the level of methods. We have used Java Source Code for the development of these metrics and we believe that it can be easily adapted to suit other object oriented programming languages as well. All the proposed metrics are numbers in the scale of 0 to 1, with 1 indicating the maximum security and 0 indicating the maximum vulnerability of the method being measured.

8.1 Access Specifier Metric (ASM)

This metric takes the value 1 if the method being measured is private and 0 otherwise. This means, public, protected and package-private methods are given the value 0. This makes sense because, as the accessibility of a method increases, its security decreases.

Let M be the method being measured. We define,

$$ASM(M) = \begin{cases} 1 & \text{if the method is private} \\ 0 & \text{otherwise} \end{cases}$$

8.2 Extensibility Metric (EM)

This method takes the value 1 if the method is final and therefore cannot be overridden. Generally, non-final methods pose a security threat in that their implementation can be changed at a lower level in the class hierarchy.
Let M be the method being measured. We define,
\[
EM(M) = \begin{cases} 
1 & \text{if the method is final} \\
0 & \text{otherwise} 
\end{cases}
\]

8.3 Sensitive Data Accesses (SDA)

The computation of this metric requires the developer to identify and mark data that are “sensitive”; in the sense that disclosure of such data is likely to lead to a security compromise. We hasten to add that, the identification of the sensitive data is essentially subjective, resting on the ability of the developer to identify such data. But we believe many programmers have a better understanding of “sensitive” data involved in their programs thereby improving the possible accuracy of such identification.

Once the developer has identified the sensitive data, the metric attempts to count the number of sensitive data accessed by the method. This includes attempts by the method to read or write such sensitive data. Formally, let \( DR=\{dr_1,dr_2,\ldots,dr_n\} \) be the set of data read by a program method M and let \( DW=\{dw_1,dw_2,\ldots,dw_n\} \) be the set of data written (updated) by M. Further let \( SDR=\{sdr_1,sdr_2,\ldots,sdr_m\} \) be the set of sensitive data read by M and \( SDW=\{sdw_1,sdw_2,\ldots,sdw_n\} \) be the set of sensitive data updated by M.

Clearly, \( SDR \subseteq DR \) and \( SDW \subseteq DW \). We define the metric,
\[
SDA(M) = 1 - \frac{|SDR| + |SDW|}{|DR| + |DW|}
\]

where
\[
|X| \text{ indicates the cardinality of the set } X.
\]

The value of this metric decreases as the method attempts to access many sensitive data, thus indicating an increased vulnerability (or a decreased security).

8.4 Native Method Calls (NMC)

Native method calls pose a serious threat to a method as such methods are beyond the control of the JVM, thus rendering the many security features provided by JVM unavailable. As the number of native method calls in a method increases, the method becomes more vulnerable. This metric makes sense only for source code developed using Java.

Let \( NM=\{nm_1,nm_2,\ldots,nm_n\} \) be the set of native method calls made by a method M and let \( S=\{s_1,s_2,\ldots,s_n\} \) be the set of statements in M. We have \( NM \subseteq S \). We define,
\[
NMC(M) = 1 - \frac{|NM|}{|S|}
\]

8.5 Methods Called Metric (MCM)

Whenever a method calls another method, the calling method becomes vulnerable to any security compromises in the called method. As a general Computer Security Principle [8] a calling method should never trust a called method. We therefore propose a metric that measures the vulnerability (or security) of a method based on the number of calls it makes to other methods.

Let \( MC=\{mc_1,mc_2,\ldots,mc_n\} \) be the set of method calls made by a method M and let \( S=\{s_1,s_2,\ldots,s_n\} \) be the set of statements in M. Clearly \( MC \subseteq S \). We define,
\[
MCM(M) = 1 - \frac{|MC|}{|S|}
\]

8.6 Validated Parameters Metric (VPM)

Whenever a method has parameters, it has to validate the parameters before attempting to use any of them. Failure to do so results In a vulnerability. A called method should never trust or have assumptions about the calling method. We propose a metric that measures the ratio of the number of parameters validated to the total number of parameters in the method. Formally, let \( P=\{p_1,p_2,\ldots,p_n\} \) be the set of parameters taken by a method M and let \( VP=\{vp_1,vp_2,\ldots,vp_n\} \) be the set of parameters validated by M. We have \( VP \subseteq P \). Now,
\[
VPM(M) = \frac{|VP|}{|P|}
\]

8.7 Logged Accesses Metric (LAM)

Whenever a method must access sensitive data, it must log the access as such a log would be of great value in the detection of any malicious activity attempted on the method. We therefore propose a metric that penalizes methods that fail to log accesses to sensitive data. This metric also, like the Sensitive Data Accesses Metric, relies on the developer marking the sensitive data in the program.

Let \( SA=\{sa_1,sa_2,\ldots,sa_n\} \) be the set of sensitive data accesses made by a method M and let \( LSA=\{lsa_1,lsa_2,\ldots,lsa_n\} \) be the set of sensitive data accesses that are logged by the method. Clearly, \( LSA \subseteq SA \). We define,
\[ LAM(M) = \frac{|LSA|}{|SA|} \]

8.8 Object Initialized Check Metric (OICM)

It is accepted principle in Secure Java Coding [9] that a method should not be coded with the assumption that the invoking object should have been initialized by a constructor. It is always possible for an attacker to construct an object using such other techniques like deserialization. So every method must have a check at the beginning to ensure that the object has been initialized. The exact way of accomplishing this is discussed in [9]. We define,

\[
OICM(M) = \begin{cases} 
1 & \text{if the method checks that} \\
& \text{the object has been} \\
& \text{initialized} \\
0 & \text{otherwise}
\end{cases}
\]

for a non-static \( M \) and

\[
OICM(M) = \begin{cases} 
1 & \text{if the method checks} \\
& \text{that the class has} \\
& \text{been initialized} \\
0 & \text{otherwise}
\end{cases}
\]

for a static \( M \).

6. Tool Implementation

We developed a tool that can scan a java source program and compute several of the metrics discussed above. The java source code and the identification of the sensitive data are input to the tool, which then computes and reports the values of the ASM, SDA, EM, NMC and MCM metrics for every method contained in the source code. The tool was implemented on the .NET Framework 2.0 using the VB.NET language on Windows XP Platform. A Screenshot of the tool is shown.

7. Case Study

To demonstrate the validity of the proposed metrics, we investigated 3 applications developed by a local software development organization in our town. Data pertaining to the attacks reported on those applications were available to us. The organization had discovered the vulnerabilities exploited by the attacks and patched them. We took the unpatched original version of the applications and computed the values for all the proposed metrics. Out of the 3 applications, 2 were developed using JSP and one using Servlets. As the number of methods in the applications was enormous, we took a representative set of 5 methods from each application, this selection was based on the significance of the methods to the application and from the filtered list 5 methods were selected at random.

For the ASM, SDA, EM, NMC and MCM metrics we used the developed tool and for the remaining we resorted to a manual static
analysis of the source code. Instead of reporting the values of all the metrics individually, we report the sum of the computed values of all the 8 metrics. The minimum value possible for such a sum will be 0 and the maximum possible value will be 8 as the values for all the proposed metrics are in the range of 0 to 1. We tabulate the results below:

**Table 1: Results for Application 1 (JSP)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Computed Metric Value</th>
<th>No. of attacks reported on the application</th>
<th>No. of attacks that exploited a vulnerability in this method</th>
<th>Percentage of attacks that exploited a vulnerability in this method</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>5.80</td>
<td>17</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>M2</td>
<td>2.01</td>
<td>17</td>
<td>6</td>
<td>35.29%</td>
</tr>
<tr>
<td>M3</td>
<td>3.97</td>
<td>17</td>
<td>3</td>
<td>17.65%</td>
</tr>
<tr>
<td>M4</td>
<td>4.02</td>
<td>17</td>
<td>3</td>
<td>17.65%</td>
</tr>
<tr>
<td>M5</td>
<td>5.62</td>
<td>17</td>
<td>1</td>
<td>5.88%</td>
</tr>
</tbody>
</table>

**Table 2: Results for Application 2 (Servlets)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Computed Metric Value</th>
<th>No. of attacks reported on the application</th>
<th>No. of attacks that exploited a vulnerability in this method</th>
<th>Percentage of attacks that exploited a vulnerability in this method</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>6.27</td>
<td>21</td>
<td>1</td>
<td>4.76%</td>
</tr>
<tr>
<td>M2</td>
<td>4.31</td>
<td>21</td>
<td>3</td>
<td>14.29%</td>
</tr>
<tr>
<td>M3</td>
<td>3.22</td>
<td>21</td>
<td>4</td>
<td>19.05%</td>
</tr>
<tr>
<td>M4</td>
<td>5.67</td>
<td>21</td>
<td>1</td>
<td>4.76%</td>
</tr>
<tr>
<td>M5</td>
<td>2.91</td>
<td>21</td>
<td>7</td>
<td>33.33%</td>
</tr>
</tbody>
</table>

**Table 3: Results for Application 3 (JSP)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Computed Metric Value</th>
<th>No. of attacks reported on the application</th>
<th>No. of attacks that exploited a vulnerability in this method</th>
<th>Percentage of attacks that exploited a vulnerability in this method</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>4.33</td>
<td>32</td>
<td>5</td>
<td>15.62%</td>
</tr>
<tr>
<td>M2</td>
<td>2.67</td>
<td>32</td>
<td>7</td>
<td>21.87%</td>
</tr>
<tr>
<td>M3</td>
<td>3.94</td>
<td>32</td>
<td>5</td>
<td>15.62%</td>
</tr>
<tr>
<td>M4</td>
<td>5.32</td>
<td>32</td>
<td>2</td>
<td>6.25%</td>
</tr>
<tr>
<td>M5</td>
<td>5.12</td>
<td>32</td>
<td>3</td>
<td>9.38%</td>
</tr>
</tbody>
</table>

### 8. Results And Discussion

For all the 3 applications, it is observed that a method with a lower value for the metric, had more attacks reported on it, than those with higher values. As we have said, metrics should be interpreted in relative terms and not in absolute terms. For instance, in the case of Application 2, both the M2 and M4 methods had the same number of attacks reported on them, though M2 has a 0.60 value higher than M4 for the metric. But this we believe, in any way does not diminish the validity of the metric as there has been a case where a metric with a lower value for the metric reporting a lower number of attacks on it.

### 9. Conclusions And Future Work

In this paper we have proposed security metrics that attempt to measure the security of software based on the source code at the method level. We have validated our claims by providing empirical results from a case study.

As a part of future work, we propose to envisage more such metrics at different levels (at the class level, at the application level). We believe that such works would be of immense help to organizations attempting to engineer “secure” software.
10. References


