# Security Guidelines: Requirements Engineering for Verifying Code Quality

Zeineb Zhioua
zeineb.zhioua@sap.com

Yves Roudier
yves.roudier@eurecom.fr

Stuart Short
stuart.short@sap.com

Rabea Boulifa Ameur
Rabea.Ameur-Boulifa
@telecom-paristech.fr

***Abstract*—The development and delivery of secure software is a challenging task, that gets even harder when the developer tries to adhere to both application and organization-specific security requirements translated into security guidelines. These guidelines serve as best practices or recommendations that help reduce application exposure to vulnerabilities, and provide hints about the application's adherence to high-level and abstract security requirements. In this paper, we present guidelines we gathered from different sources, and we highlight the main issues related to the interpretation and application of those guidelines. We present a first attempt to classify the requirements with the objective of identifying the analysis that should be performed to verify the adherence of the developed software to each of the categories.**

***Keywords- Security Guidelines, Development Lifecycle, Information flow***

## I. Introduction

With the aim of protecting the infrastructure and sensitive data of their own as well as of their customers, organizations and companies define non-functional security requirements to be applied by software developers. Defined requirements are generally abstract and high-level, which requires extra effort to interpret, apply and verify the adherence to. In practice, companies create review processes and conduct audit sessions [16] comprising a wide range of experiments to verify the compliance with general and security related requirements. However, manual reviews can be time-consuming and costly to the companies in terms of resources, and they can fall short in detecting requirements violations. From a developer perspective, interpreting and applying security requirements is not a trivial task, as those requirements, translated into security guidelines, are often written in an informal style, use ambiguous language, and require domain expertise to interpret them. The idea behind this paper is not to propose a model-driven engineering approach offering methodologies for designing secure systems [24], but we aim at bringing the verification of security guidelines in early stages of the development lifecycle, and provide assistance and guidance to the developer through the development phase to ensure the quality of his developed software with respect to security. We propose a first step towards closing this gap, consisting at arranging the guidelines into categories with the objective of preparing the ground for the analysis approach to be carried out for each category. In order to define the categories, we conducted a deep analysis on guidelines coming from diverse sources. The analysis led to the classification of a preliminary set of 17 guidelines, that will be used for creating models for formal verification. The paper flow is as follows: Section 2 provides the motivation behind this paper. We introduce the security guidelines in Section 3 and propose a classification of the guidelines with respect to the identified criteria. This presentation is then followed by a discussion in Section 4. In Section 5, we provide existing approaches that dealt with guidelines specification. Section 6 concludes the paper and discusses possible directions to verify the compliance with security guidelines.

## II. Motivation

With the objective of illustrating further the problems described above, we provide a sample code (Fig. 1) in which user credentials are provided as input. On the other hand, we want to verify the adherence of the developed code with a specific guideline that recommends to hash passwords before logging them (denoted IDS03-J in [5]). The developer assigned the input password to the password attribute of an object *user* (line 99). In the lines 105 and 106, the developer instantiates the *Sanitize* and the *Log* threads (Threads implementation is depicted in Fig. 2). The *Sanitize* thread eliminates the suspicious characters from a given data. In our example, the data is the user password provided as input. The *Log* thread, on the other hand, performs the logging operation. Later in the program, the developer runs the thread *Sanitize* followed by the thread *Log* (lines 108 and 109). From a pure control flow angle, that is from methods invocation perspective, the guideline is met, as the password provided as input was first sanitized (line 108) before being logged (line 109). However, in the presence of multiple threads, the control flow of the program can be altered, and the sensitive data "'password"' can be modified in inconsistent ways, as the logging operation might occur before the sanitization, and this will lead to the password leakage, hence, to the violation of the guideline. In this example, detecting the violation of the guideline is not trivial; there is a need for a reliable analysis that should detect this violation. Nowadays, formal methods, in particular formal verification are increasingly being used to enforce security and safety of programs. Formal specification of the guidelines is not in the scope of this paper.

## III. Security Guidelines

Organizations and companies define non-functional security requirements to be applied by software developers, and those requirements are generally abstract and high-level. Security

```
92          BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
93          System.out.println("User name : ");
94          System.out.println("Password : ");
95
96          // input
97          try {
98              user.setUsername(reader.readLine());
99              user.setPassword(reader.readLine());
100         }
101         catch(IOException e) {
102             e.printStackTrace();
103             }
104
105         Thread sanitize = new Sanitize(user.getPassword());
106         Thread log = new Log(user.getPassword());
107
108         sanitize.start();
109         log.start();
```

Fig. 1. Sample code 1

```
50  public static class Sanitize extends Thread {
51      public Sanitize(String data){
52          if ((data.charAt(0) =='<') || (data.contains("<script"))
53              || (data.contains("<"))) {
54          data= data.substring(data.indexOf("<"), data.length());
55          }
56      }
57
58      public void run() {
59          System.out.println("sanitize");
60      }
61  }
62
63  public static class Log extends Thread {
64      public Log(String data) {
65          String logMessage;
66          final Logger logger = Logger.getLogger("User");
67          logMessage = "password = " + data;
68          logger.log(Level.INFO, logMessage);
69      }
70
71      public void run() {
72          System.out.println("log");
73      }
74  }
```

Fig. 2. Sanitize and Log threads

requirements such as confidentiality and integrity are abstract, and their application requires defining explicit guidelines to be followed in order to fulfill the requirements. Security guidelines describe bad as well as good programming practices that can provide guidance and support to the developer in ensuring the quality of his developed software with respect to the security aspect, and hence, to reduce the program exposure to vulnerabilities when delivered and running on the customer platform (on premise or in the cloud). Bad programming practices define the negative code patterns to be avoided, and that can lead to exploitable vulnerabilities, while good programming practices represent the recommended positive code patterns to be applied on the code. However, guidelines are usually presented in an informal and unstructured way. In the OWASP Secure Coding Practices guide[21], a set of security guidelines are presented in a checklist format arranged into classes, like Database Security, Communication Security, etc. The listed programming practices are general, in a sense that they are not tied to a specific programming language. The same source, OWASP [20] introduces the **Cryptographic Storage Cheat sheet** that provides a set of guidelines to be followed in order to protect data at rest. Another programming practices guide we can consider for instance is the CERT Oracle Coding Standard for Java [14]; for each guideline, the authors provide a detailed textual explanation. For most, there are also provided examples of compliant and non-compliant sample codes in addition to the description. The Juliet Test Suite [19] is created by the National Security Agency (NSA), and proposes a set of test cases covering multiple programming languages including Java. The provided test cases are arranged into categories with respect to the flow type; control or data, and every test case targets one specific vulnerability or weakness, referring to known entries in the Common Weakness Enumeration dictionary (CWE) [1].

The definition of security guidelines also stands in another scenario: software marketplaces. In this scenario, the main stakeholders are the service provider or the developer, the marketplace operator and the end-user. Software Marketplaces use different approaches in order to prevent malicious applications from being advertised in their official stores. Submitted applications generally go through a verification of their compliance with the marketplace security requirements. On the one hand, it is important for a service provider to be aware of domain-specific requirements as failure to be accepted in the marketplace and subsequent application rewrites may be costly, time-consuming, and affect the organization's reputation. On the other hand, the end-user wants to be reassured that their demands are respected in terms of security and privacy. However, this process is neither always transparent nor understandable to the service provider as well as to the end-user. For instance, Apple App Store performs a rigorous vetting process that comprises in addition to security and privacy checks, verifies also the adherence to the Apple App Store guidelines [1], that serve as a guide to developers through the development of their applications and to enhance their chances of getting their applications pass the approval process. From service providers/developers perspective, the understanding and interpretation of guidelines is not trivial, as

[1]https://cwe.mitre.org/index.html

there is no formalization that exposes the necessary program instructions for each guideline, or that explains how to apply them correctly in the software.

Before proceeding to the classification, effort was undertaken to develop the criteria serving as basis to our categorization. One trivial criterion is the control flow, specifying the sequence of operations that should be performed for a software to be compliant. Another criterion consists in considering data and how it should be processed. Different guidelines require some data typing rules, specifying the data types that should be used in a given context. The fourth criterion we have considered is the semantic aspect. This criterion means that there is a need to study the meaning of linguistic expressions for the evaluation of the guideline's key words. The interpretation of semantic guidelines is not straightforward, and requires expertise to analyze, apply and verify them.

The main objective behind this classification is to prepare the ground for expressing these guidelines in a formal language and to reason about their satisfiability. Control, data and typing guidelines can be translated automatically into formal language. However, for those involving the semantic aspect, the formalization cannot be carried out automatically and should be led by a security expert who extracts the linguistic meaning and relate it to the guideline context.

Let's now consider some real-world guidelines [2] as introduced in different sources such as CERT [14], Apple App Store Developer Guide [1], Juliet Test Cases[19] and OWASP[20]. Each guideline is denoted by a unique code attributed by the issuing organization.

As a first step, we will organize the guidelines into groups with respect to the type of verification they consider. For example, a category of guidelines can be more focused on validating data provided as input. We can also consider another category for secret security, that is data whose integrity violation can result in a loss of confidentiality of sensitive information. Sensitive information security is another category we define, and consists at regrouping the guidelines for the sensitive information, such as user's private information. We have also depicted categories that deal with specific methods invocation and execution environment security. For each guideline, we provide explanation, comments and observations.

### A. Input validation

This category gathers the guidelines that perform the validation of data provided as input. Validation consists in ensuring that input data belong to the expected input domain.

*a) IDS01-J:**Normalize strings before validating them**:* [11] This guideline recommends that input strings should be normalized before being validated. Normalization is according to this guideline a crucial operation as the same string can have multiple representations depending on the used Unicode. However, neither the normalization nor the validation are easy to interpret and to apply. Does the validation mean sanitization?

*b) IDS03-J:**Do not log unsanitized user input**:* [5] This guideline describes the recommended behavior when dealing with user input that should be logged. Provided user input should be sanitized before being logged, however, sanitization can be performed through multiple manners, such as the elimination of suspicious characters. In the guideline description, the implementation of the sanitization operation is not specified.

*c) IDS06-J:**Exclude unsanitized user input from format strings**:* [8]

The guideline recommends not to include untrusted data in a format string, as this may result in information leakage. One ambiguity in the guideline description is the notion of **untrusted data**. From a developer perspective, the operation of "'excluding"' untrusted data is not trivial; is it only about not including it in format string? Hence, this guideline can be formulated as follows: sanitize user input before including it in format string.

*d) IDS07-J:**Sanitize untrusted data passed to the Runtime.exec() method**:* [13]

This guidelines proposes a way to reduce the program exposure to command and argument injection vulnerabilities, through the sanitization of untrusted data included in a specific Java method *Runtime.exec()* that allows to run an external program. It is pointed out that suspicious arguments passed to this method may expose the program to the command injection attack. Despite the detailed description, applying the guideline is not trivial, due to the unclarity of the untrusted data notion, even though there is an attempt to clarify it and put in the context of *trust boundary*.

*e) IDS08-J:**Sanitize untrusted data included in a regular expression**:* [12]

In this guideline, it is recommended to sanitize data passed to a regex in order to prevent malicious attacks such as regex injection, information leak or DOS attacks. If unsanitized input is included in a regex, this might modify the original regex that will be changed and will no longer perform the original desired verification. The developer is faced with the ambiguous notion of *untrusted data*. What would determine if the data is trusted or not? Does this assume that all the data coming from external sources (user input, consumption of a web service, etc.) is untrusted?

*f) 124683:**CWE 129: Improper Validation of Array Index**:* [18]

The test case 124683 references a weakness in the CWE (Common Weakness Enumeration database) that points out the risk of not validating the array index[3]. It is recommended to check that the array index is within the correct range of values for the array, but it is not specified how to properly perform this verification.

### B. Method declaration and invocation

This category is focused on guidelines that provide useful hints to ensure the program safety when invoking specific Java methods.

[2]https://www.securecoding.cert.org/

[3]https://cwe.mitre.org/data/definitions/129.html

*g) MET03-J:**Methods that perform a security check must be declared private or final**:* [9] This guideline recommends to declare methods that perform security checks as private or final to make sure they cannot be overridden. However, the notion of *security checks* can be subject to different interpretations; is it authentication? encryption verification? As a reader can notice, the understanding and implementation of this guideline are not intuitive.

*h) MET53-J:**Ensure that the clone() method calls super.call()**:* This guideline specifies that for a class that implements the *clone()* method, it should explicitly invoke the *super.clone()* method. Otherwise, the types of the cloned object and the original one can be different.

*i) MET56-J:**Do not use Object.equals() to compare cryptographic keys**:* [7] The guideline recommends avoiding the invocation of the method Object.equals() to compare cryptographic keys. The motivation behind this guideline is that *Object.equals()* method is not applicable to composite objects, such as cryptographic keys, and may return false even when the keys have exactly the same value.

*j) EXP02-J:**Do not use the Object.equals() method to compare two arrays**:* [6]

This guideline specifies the proper method to be invoked for the comparison of the Java Array type.

### C. Secret security

We express through this category the need to ensure the integrity of specific data whose violation can have negative impact on confidentiality of dependent data. For example, the violation of password integrity can violate authentication, same for the violation of encryption keys secrecy, this will result in violating confidentiality of encrypted data, and the encryption operation can no longer be reliable for ensuring confidentiality.

*k) MSC62-J:**Store passwords using a hash function**:* [15]

This guideline specifies the technique to be used in order to ensure the secrecy of passwords and limit their exposure. The hash operation applied on passwords which are the sensitive information in this guideline, will result in an undecryptable data that can propagate to a public output without having any risk on its secrecy. From this perspective, the guideline somehow indicates a technique for declassifying the security level of sensitive information such as passwords. However, the identification of a specific variable or data as password requires semantic inference rules and a rich knowledge base.

*l) **Store unencrypted keys away from the encrypted data**:* [20] This guideline recommends not to store encrypted together with the encryption key, as this operation can result in a compromise for both the sensitive data and the encryption keys. However, encryption keys can be declared as byte arrays with insignificant names, which makes their identification as secret and sensitive data very difficult.

### D. Sensitive information security

In this category, we collected the guidelines that deal with sensitive data, that include user's private information, passwords, credit card numbers, etc.

*m) MSC03-J:**Never hard code sensitive information**:* [10]

In this guideline, it is recommended not to expose sensitive information on plain text at the code level. This guideline is not about the behavior of the program, but it is more about the code, meaning class files. The motivation behind it is the risk of exposure of sensitive data if an intruder gets access to the class files or decompile the byte code, then he can easily discover sensitive data.

*n) IDS15-J:**Do not allow sensitive information to leak outside a trust boundary**:* [4] This guideline is considered as a stub, in a sense that it is generic, and can be instantiated through different mechanisms, like for instance preventing catching an exception that exposes sensitive data, or forbidding the storage of sensitive data on external storage, etc. The notion of trust boundary can lead to misinterpretation, hence to improper implementation. From a developer perspective, it is tough to identify all the sensitive information in his program, the complexity of this operation increases with the complexity and the size of the program.

*o) 5.1.2(i) **Apps cannot use or transmit someones personal data without first obtaining their permission and providing access to information about how and where the data will be used.**:* [1]

This guideline explains that applications requiring access to user's data should explicitly request the access to private data and should also inform the user about where the data will be processed and to which end. From a developer point of view, the identification of private information cannot be easily carried out. This operation requires an advanced semantic knowledge base covering the main naming conventions of personal information.. The developer is faced with another lack of clarity concerning the access to data; the access is not explicitly defined: is it read, write, modify, delete? There is a lack of precision about the purpose of the access, and also about how to inform the user about where and how the data will be processed. On top of that, the operation of granting or revoking a requested permission occurs at install time or even run-time for some specific permissions, hence, verifying whether the application was granted the permission or not cannot be checked statically. The problems described are also faced by the tester or the reviewer of the application.

*p) 5.1.2(ii) **If your app doesnt include significant account-based features, let people use it without a log-in. Apps may not require users to enter personal information to function, except when directly relevant to the core functionality of the app or required by law..**:* [1]

The guideline explains that applications requesting access to private information will be rejected. We are faced with the some ambiguities discussed for the previous guideline.

### E. Execution environment security

This category is focused on guidelines that aim at ensuring the security of the execution environment, They are neither control flow nor data flow guidelines.

*q) OBJ10-J:**Do not use public static nonfinal fields**:* This guideline recommends not to declare nonfinal fields as public static, for the risk of exposure to improper change of their values when there is a concurrent access to their content, especially in the presence of multiple threads, where the concurrent access to nonfinal public static fields can result in an inconsistent modification of the content.

### *F. Summary and classification of security guidelines*

By arranging the security guidelines into categories with respect to the flow aspect; control or data or a some combination thereof, we will be able to have concrete understanding on the analysis type that will be performed on the program in order to verify the adherence or not to the given guideline. We have also considered some other categories that are neither control nor data, but more about the data typing or semantic aspect of the guideline. The classification of the guidelines is presented in Table 1. Going back to our motivating example in Section 2, we notice that the nature of the guideline to be verified is a combination between both control and data flows. Hence, in order to validate it, we need to perform an information flow analysis to track how the password data propagates through the program, taking into account the implicit information flows that can originate from the occurrence of multiple threads.

## IV. Discussion

Security guidelines are mainly meant to simplify the developer job in enhancing the software quality from security perspective. However, if we take a closer look at the guidelines presented above, we will notice that their understanding and implementation are not trivial to the developer. Considering only the events or actions from pure control flow angle is not sufficient for guidelines that are highly dependent on the information propagation through a program, such is the case for the guidelines IDS03-J - IDS15-J. Hence, considering information flow and integrating them into the formal specification of security guidelines becomes imperative.

When interpreting the guidelines, the developer is faced with one other problem related to the semantic interpretation of key-words such as the notion of sensitive data, trusted/untrusted data, trust boundary, security check, improper validation, etc. From this perspective, the developer should be able to identify the sensitive variables or data in his developed software, the problem gets even harder when it is the tester who has to perform this identification on a developed program. It is also the case for software marketplaces when performing the approval process.
In the guidelines presented above, we notice the redundancy of the notion of trusted or untrusted data, which is ambiguous to interpret, to understand and to apply on the source code level. In the CERT source, the authors provide a glossary to explain further the different key-words and terms used in the guidelines. The *untrusted data* is defined as *Data originating from outside of a trust boundary*, which leads to uncertainty with respect to the identification and determination of trust boundary; does it refer to the system input/output? It is not trivial to determine the perimeter of a program, for example in the context of a distributed application, how to determine the system boundaries?

The validation operation is used in multiple guidelines, for example in IDS01-J and 124683. As one can notice, the validation of array index is not the same validation operation from program instructions point of view.

Another element caught our attention; the guideline or the family of guidelines that recommends to encrypt or hash the sensitive information. This operation always known as *declassification* is widely known in the information flow vocabulary [22]. Declassification operation can be seen as controlled release of sensitive information using given mechanisms and techniques such as encryption, hashing, obfuscation, etc.

Some rules do not have the same permissibility level, in a sense if they are applied at the same time, this might lead to ambiguities. For instance, the guidelines IDS06-J and IDS03-J; the first rule forbids user input from format string, while the second requires to sanitize user input. This brings to the table the necessity of considering the relationships and dependencies between guidelines.

It is important to note that all the specified categories except *Secret security* are focused on data confidentiality, and somehow miss the integrity which is according to Biba [3] the dual of confidentiality. Integrity unlike confidentiality, can be violated without any interaction with components external to the system. This is the main major behind considering *Secret security* category as an integral part of our classification.

## V. Related Work

The classification of security guidelines has also been addressed in the literature. For instance, HP Fortify[17]carries out the classification of the analysis rules depending on the vulnerability type, and makes use of a specific analyzer for each category: Control, Data, Semantic, Structural and Configuration. However, the analysis approach is not transparent, neither is the specification of analysis rules. Apple App Store proposes a set of guidelines that should be met by the application developers/providers. The guidelines are arranged into categories with respect to their context (safety, business, legal, etc.). Apple reported in their reply to the FCC questionnaire [2] that for an application to be accepted, it has to undergo a vetting process comprising different verifications such as the compliance with the guidelines. To achieve this, the application is tested (automatically and manually) by more than one reviewer. John Wilander and Pia Fak[23] identified verification categories depending on the vulnerabilities types. The authors proposed GraphMatch tool that considers examples of security properties covering both positive and negative ones, that meet good and bad programming practices. GraphMatch is more focused on control-flow security properties and mainly on the order and sequence of instructions.

| ID | Security guideline | Validation approach | | | |
|---|---|---|---|---|---|
| | | Control | Data | Typing | Semantic |
| | **Input validation** | | | | |
| IDS01-J | Normalize strings before validating them | X | X | | |
| IDS03-J | Do not log unsanitized user input | X | X | | |
| IDS06-J | Exclude unsanitized user input from format strings | X | X | X | |
| IDS07-J | Sanitize untrusted data passed to the Runtime.exec() method | X | X | | |
| IDS08-J | Sanitize untrusted data included in a regular expression | X | X | | |
| 124683 | CWE 129: Improper Validation of Array Index | X | X | | |
| | **Method declaration and invocation** | | | | |
| MET03-J | Methods that perform a security check must be declared private or final | X | | X | X |
| MET53-J | Ensure that the clone() method calls super.clone() | X | | | |
| MET56-J | Do not use Object.equals() to compare cryptographic keys | X | | X | X |
| EXP02-J | Do not use the Object.equals() method to compare two arrays | X | | X | |
| | **Secret security** | | | | |
| MSC62-J | Store passwords using a hash function | X | X | X | X |
| | Store unencrypted keys away from the encrypted data | X | X | | X |
| | **Sensitive information security** | | | | |
| MSC03-J | Never hard code sensitive information | X | X | | X |
| IDS15-J | Do not allow sensitive information to leak outside a trust boundary | X | X | X | X |
| 5.1.2(i) | Apps cannot transmit data about a user without obtaining the user's prior permission and providing the user with access to information about how and where the data will be used | X | X | | X |
| 5.1.2(ii) | Apps that require users to share personal information, such as email address and date of birth, in order to function will be rejected | X | X | | X |
| | **Execution environment security** | | | | |
| OBJ10-J | Do not use public static nonfinal fields | | | X | |

TABLE I
SECURITY GUIDELINES ARRANGED INTO CATEGORIES

## VI. Conclusion and Future Work

In this paper, we proposed a first attempt at classifying the security guidelines gathered from different sources into categories. We pointed out another key issue with the guidelines, consisting in considering only *closed* systems, and how they fall short in treating distributed systems and the guidelines complexity they induce. The complexity can be perceived from the difficulty of defining the system boundaries for distributed and 3-tier systems. We are working towards closing the gap between the informal aspect of the guidelines, and their implementation and application through a formalism that allows to strip away ambiguities. We aim to express these guidelines in a formal language and to reason about their satisfiability. Concerning those related to the first three criteria, that is control, data and typing, they should be translated automatically into formal language. However, for those involving the semantic criterion, the formalization should be led by an expert aware of the software security domain, as semantic aspect is highly tied to the context. We are currently developing a code analysis tool for the automatic or semi-automatic verification of such formalized security guidelines

## References


[1] Apple. App store review guidelines.
[2] Apple. Apple answers the fcc's questions.
[3] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.
[4] CERT. Do not allow sensitive information to leak outside a trust boundary.
[5] CERT. Do not log unsanitized user input.
[6] CERT. Do not use the object.equals() method to compare two arrays.
[7] CERT. Ensure that the clone() method calls super.clone().
[8] CERT. Exclude unsanitized user input from format strings.
[9] CERT. Methods that perform a security check must be declared private or final.
[10] CERT. Never hard code sensitive information.
[11] CERT. Normalize strings before validating them.
[12] CERT. Sanitize untrusted data included in a regular expression.
[13] CERT. Sanitize untrusted data passed to the runtime.exec() method.
[14] CERT. Sei cert oracle coding standard for java.
[15] CERT. Store passwords using a hash function.
[16] Facebook Ireland Ltd Data Protection Commissioner. Report of re-audit. September 2012.
[17] HP Fortify. Hp fortify static code analyzer:user guide.
[18] NIST. Cwe: 129 improper validation of array index.
[19] NSA. Juliet test suite.
[20] OWASP. Cryptographic storage cheat sheet.
[21] OWASP. Owasp secure coding practices quick reference guide.
[22] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
[23] John Wilander and Pia Fak. Pattern matching security properties of code using dependence graphs.
[24] Roudier Yves and Apvrille Ludovic. Sysml-sec: A model driven approach for designing safe and secure systems. *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference*, pages 655–664, February 2015.