Case Study: Applying common software security guidelines to improve program robustness

Stefan Thorén

toStefanT@hotmail.com

CompSci 725, Software Security – Term project

Computer Science Department, The University of Auckland

Auckland, New Zealand, June 10, 2002

Abstract

This paper describes a case study where security objectives are applied to a Java code package that is suitable as a framework. However, to become a product the package requires an extra layer of robustness. To evaluate the security objectives a security audit is performed that is based on common software security guidelines and aims to identify vulnerabilities and increase the knowledge of possible threats. The inspection uncovered some program flaws, as a result by developing without security objectives and identified security rules that were applicable for a local Java application.

Keywords: program flaws, security objectives, Java, XML, program robustness, protection mechanisms, security guidelines, code security.

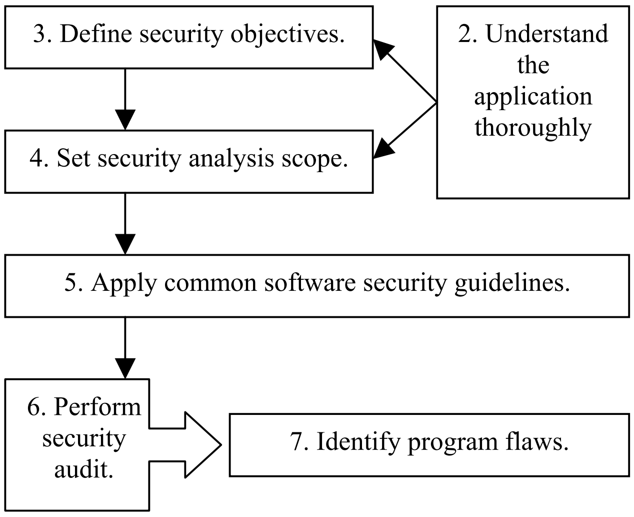
1. Introduction

Java has always shown promise by being available anywhere, anytime, on any device and is nowadays also integrating with new software inventions such as web services, XML[1], mobile and embedded applications. The amount of code developed using this multi-platform technology is huge and demand high security requirements. Various tools and techniques exist that can help to produce robust, reliable and secure software, however there are probably as many methods for breaking in to code. It may seem like "maximum" security is the only solution, but because of the overwhelming complexity this means for an application the security objectives might be abandoned. This highlight issues such as; whether it is of any value to add security mechanisms; whether vulnerabilities can be

[1] XML=Extensible Markup Language, allow structuring of data within descriptive tags (e.g. <tagName></tagName>).

addressed efficiently in the design phase, before coding even starts; and how security objectives should be implemented at the code level. Sometimes, security experts give advice and guidelines of what to do to implement security, however these suggestions are often addressing security at a conceptual level [2,3] and also to cover the entire security scope including networking and operating systems. The opposite approach has also been presented, as in McGraw's and Felten's article "Twelve Rules for Developing more Secure Java Code", which offer short, concise security rules at code level for developers to follow [1]. This might seem, from the developer's point of view, to spare one from the need of thorough knowledge of the underlying security threats. However, it is important to clarify the hidden vulnerabilities and threats, otherwise the developer will not be able to realize why and against what protection is added and consequently will have difficulties to tackle the security objectives appropriately.

It is hard to write secure code without appropriate security objectives while an application is being developed. This paper investigates the implications of adding security objectives and what tradeoffs should be forced to face early in developing a local Java application. The code package, that is subject for investigation, forms a suitable plug-in framework for generating reports based on XML. Also, a sub goal of this exercise is to separate and generalize the code package from its original application as well as reduce dependencies to third party software (external untrusted libraries), towards a robust and secure product.


3. Define security objectives.
2. Understand the application thoroughly
4. Set security analysis scope.
5. Apply common software security guidelines.
6. Perform security audit.
7. Identify program flaws.
Figure 1. Approach for this paper.

Referring to Figure 1, *Section 2* introduces the application, to establish an overall understanding of the target of the security audit. *Section 3* defines the security objectives using Pfleeger's classification of threats and security goals [1]. *Section 4* set the analysis scope by investigating how program flaws can occur, when they might be introduced and thereby identify some possible vulnerabilities. *Section 5* gathers common software security guidelines to learn from previous experiences and to obtain expertise opinions of how secure programs should be developed. *Section 6* executes the security audit and the results are detailed in *Section 7*. The concluding discussion ventilates the decisions to take whether the architecture shall be changed or API improvements need to be done. The identified program security flaws are summarized and reusable parts of this project are promoted.

2. Understanding the application

2.1. Functionality

The Xerpt is a plug-in framework for generating reports, originally called Report Manager but was renamed during this project towards a more robust a secure product. The name “Xerpt” lend the meaning of the similar spelled word “excerpt” which means “extract, selection, piece”. Also, the acronym “Xerpt” stands for “XML based extraction of data for reports, printouts and further transformations”. It uses XML to represent data based on tags defined in a DTD[2] file and an XSL[3] style sheet to transform the data.

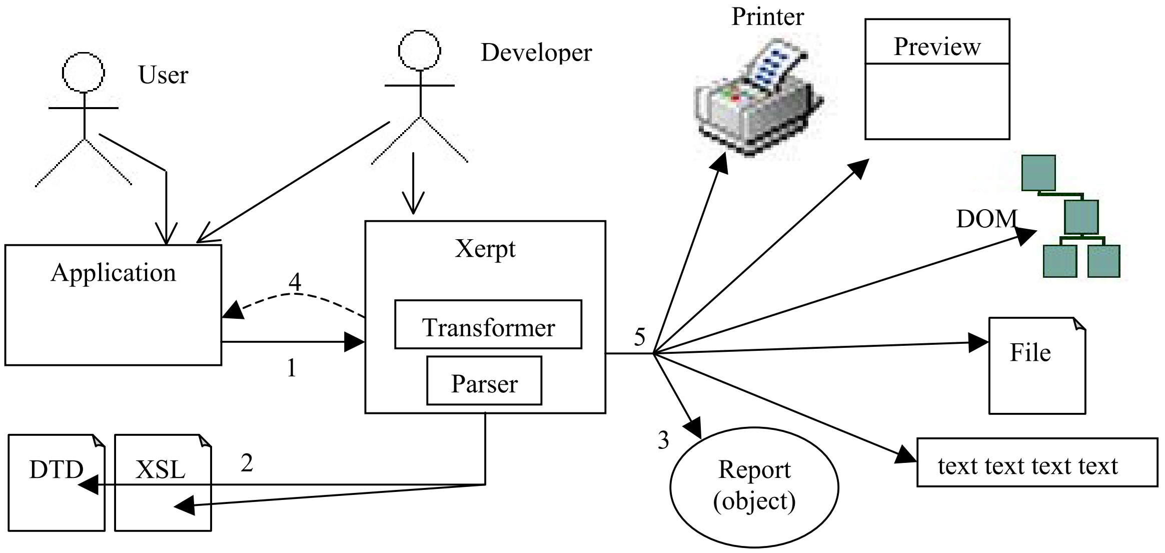


Figure 2. Conceptual overview of how report generation works with Exerpt.

The package also contains generic functionality that can be used separately: XML parsing and transformation; viewing HTML or text; show a "to file" dialog with default file names and file filters; print HTML, text, file, Swing[4] or Awt[5] components (take care of rendering multiple pages).

Figure 2 depicts the terminology used throughout this paper:

- User – person that uses the Application and the framework (do preview, print or save to file).
- Developer – person that implements Reports based on the framework.
- Application – system (in Java) that uses the framework to generate Reports.
- Xerpt – the Report generator (also referred to as “plug-in” or “framework”).
- Report – the main entity that needs to be customized for the Application domain.

[2] DTD=Document Type Definition, defines allowed tags for an XML document.
[3] XSL=XML Style sheet Language, recursive XML templates that formats the found tags in an XML document.
[4] Swing=Java user interface package with components for e.g. trees, lists, tables, buttons, menus etcetera.

- DTD – the Developer define domain specific Report tags and their relations in this file.
- XSL – the Developer define style sheets for the Report output format according to the DTD.
- Transformer – transforms an XML source with an XSL style sheet.
- Parser – parses the supplied XML source to a DOM[6] structure.
- Printer, Preview, DOM, File, Text - output formats that can be generated (also Report). Preview is a dialog that Xerpt provides that can show HTML or text.

Following the numbers in Figure 2, the description of a system using the terminology is:

1. An Application uses Xerpt to generate Reports with an XML Parser and an XSL Transformer, see references [6] for better understanding of how XML processing and transformation works.
2. The Report can be customized according to the Application domain terminology by providing a DTD file and an XSL style sheet to Xerpt (can load them from a file or within a JAR[7] file).
3. The class that implements a Report must use the tag definitions from the DTD to gather different kinds of data e.g. text, XML strings or DOM structures. The data is converted and arranged in Report “appropriate format” using factory methods and interfaces defined.
4. When a Report is generated the Xerpt fetch the data from the Application and build an XML document (DOM structure) for the source data. The framework verifies that the source XML data comply with the tag definitions allowed in the DTD file.
5. The Transformer transforms the source data to the specified output format and send it to different destinations which can be: to a printer, to the screen, as a DOM structure, to a file, to a string and to other reports (which can be regenerated according to another structure).

2.2. Preparations

The following is the suggested preparations when using Xerpt:

- Decide Report content (list data to include), layout (sketch) and the order of Report parts.
- Establish a test environment (e.g. XMLSpy[8]) to be able to develop a sample source XML data structure, the style sheet and the DTD.
- Define a DTD file with the tags for the Report source data and base an XSL style sheet on it.

---

[5] Awt=Abstract Windowing Toolkit, the predecessor user interface package before Swing was released.
[6] DOM=Document Object Model, XML structure in memory.
[7] JAR=Java Archive file, a compressed file which normally contains class libraries and resource files for Java applications.

- Include the tested and working DTD and XSL files in a JAR file to be able to load them.

- Include an XML parser and an XML transformer in the class path (e.g. Xerces[9] and Xalan[10]).

2.3. Architecture

The classes and their dependencies for: print utilities, the viewing mechanisms, the exception mechanisms and the Report generation process are portrayed in Figure 3.

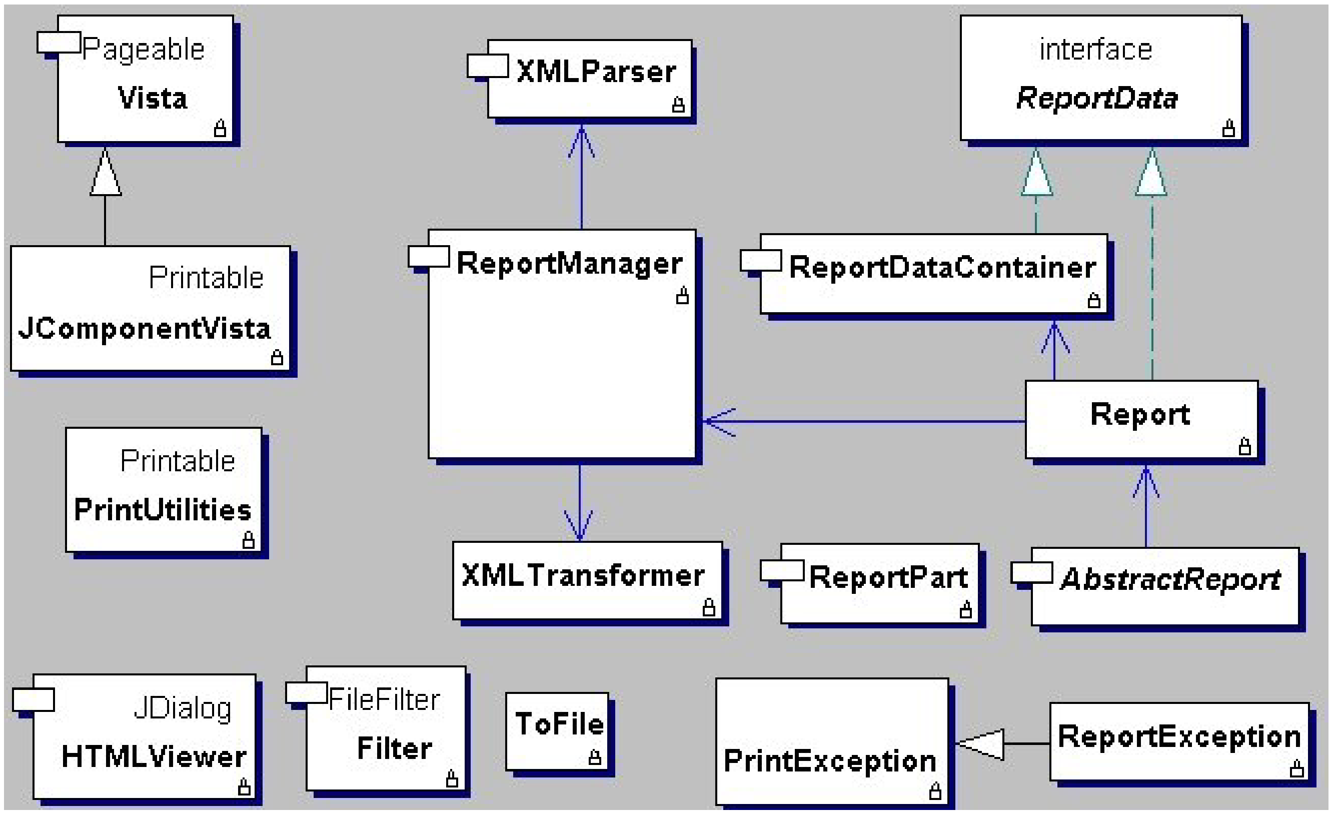


Figure 3. The Xerpt (Report Manager) package class diagram overview.

2.4. How to implement a Report

Before designing the Report for the source data one must consider the following:

- Decide conversion strategy by investigating the classes in the Application that currently includes the source information and might need to be modified. Either implement the `ReportData` interface (when allowed to change the source code) in the data source classes or

[8] XMLSpy=Tool for development of XML and XSL files, includes parsing and transformation mechanisms.
[9] Xerces=Open source project, a Java library for processing XML information according to the XML standard.
[10] Xalan=Open source project, a Java library for transforming XML information with XSL style sheets according to DTDs.

gather and tag up the data manually by creating `ReportPart` objects (when disallowed to change the source code) with the `ReportDataContainer`.

- Decide if the Report shall be:
    - *Dynamic*, once the Report is configured with the objects that hold the data in the Application can be queried and the Report becomes regenerated on the fly.
    - Or *static*, the data inserted in the Report on creation is not changed.
- Determine if sub Reports are needed, e.g. if part of the data is in the wrong format, is completely isolated or need further processing before it can be added to the source XML.
- Decide order of the parts in the Report.

When to implement a Report the following steps need to be performed:

1. Subclass `AbstractReport`, define the custom tags needed for the Report, implement the abstract methods to get the path to the style sheet and the rule set. Most work will then be to build the data by implementing the abstract method `buildData`.
2. Prepare the Report and decide if it shall be dynamic, what rule set should be valid for the tags, what style sheet to use, which tag is the root tag and where to the report shall be generated (and set default file name if destination is to file).
3. Build parts from the objects that contain the data by implementing the `ReportData` interface or create `ReportPart` objects and tag the data manually.
4. Gather data in a `ReportDataContainer` that provides different add methods (some methods enforce restrictions for dynamic mode) to be able to add data from different sources (e.g. text, XML, DOM structure or even another Report).
5. After the destination is set it is possible to generate the Report.

Figure 4 shows the main classes involved when producing a report. The classes `AbstractReport`, `ReportPart`, `ReportData` and `ReportDataContainer` are the *template classes* to be used from the Application to prepare a Report.

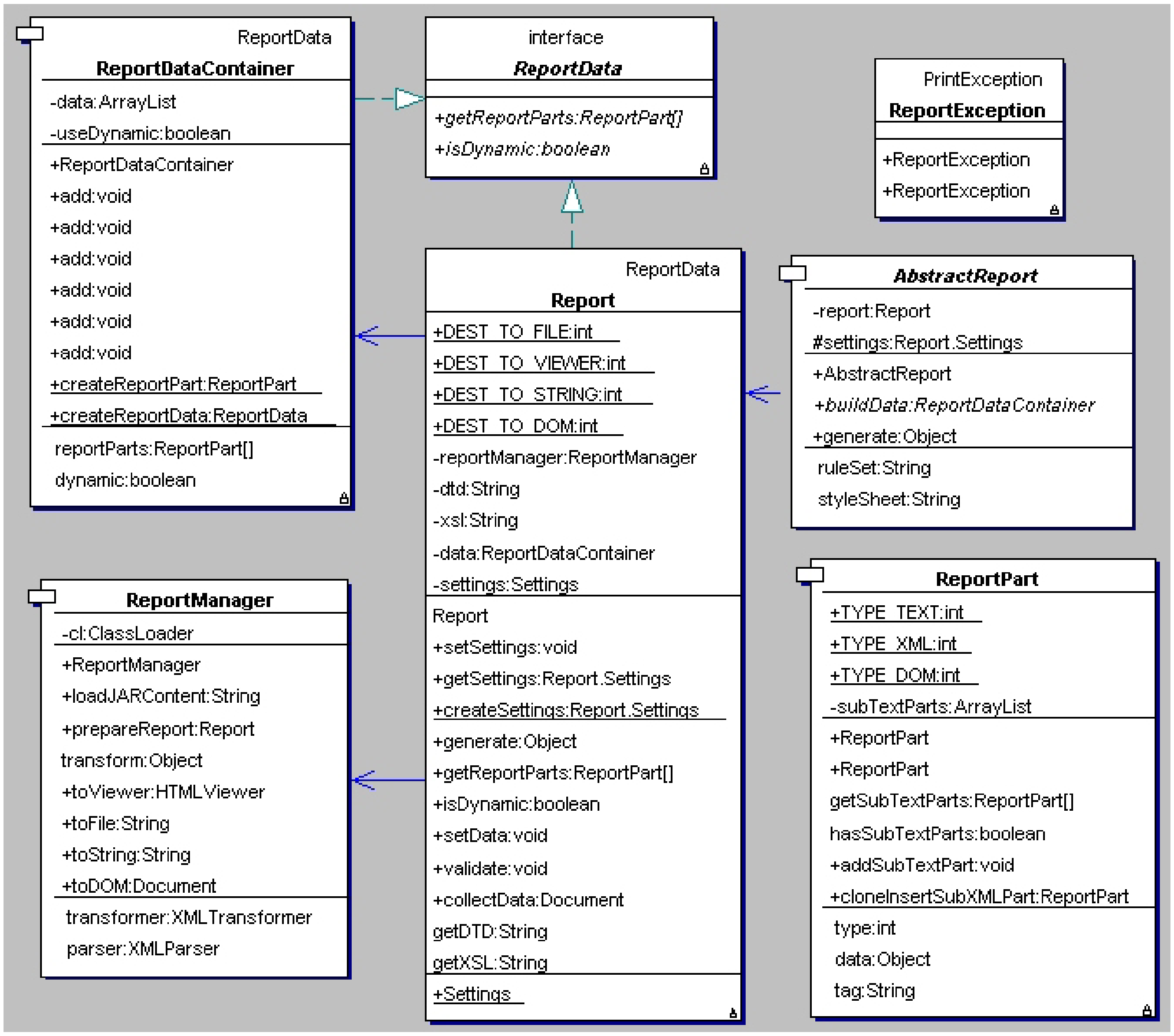


Figure 4. Classes to use when implementing a Report.

3. Defining the security objectives

Pfleeger defines, in his book “Security in Computing”, that the main goals to ensure computer security are to maintain the system characteristics: Confidentiality, Integrity and Availability (CIA) [3]. Also, he encourages to think about the persons and threats involved, attempting to break the computer security. This section uses his principles as a guideline to define the security objectives for Xerpt, aiming to clarify what shall and shall not be possible to do in the framework and evaluate them (“shall” = provide and “shall not” = prevent) separately.

The persons involved are a User or Developer that wants to use the framework without malicious intentions or possibly, but less likely, an attacker that wants to exploit the information in the

Application that might have many security protection mechanisms, but the plug-in might open up a weakness through which the information can be intercepted.

Note: The security audit checklist in the Appendix refers to the labeled lists (e.g. A, B, C) in the other sections of this document, beginning with A) beneath.

A) The framework design seeks the following security goals (according to CIA) and shall **provide**:

1. Provide an interface trough some well-defined and meaningful public classes, limit functions provided by the framework as much as possible to spare the Application from the complexity, reduce the constraints imposed on the Application (confidentiality, integrity).
2. Throw all errors back to the Application and end operations "nicely" to keep the system internally consistent (integrity).
3. The XSL style sheets, the DTD structure definition file or data that follows the DTD tags must be possible to be provided outside the framework and be configured properly (integrity).
4. Reduce printing or generation time, to not let someone wait unnecessary time (availability).
5. Have a careful and robust implementation that can stand some level of unintentional or erroneous usage without becoming unusable (availability).

B) Following Pfleeger's classification of threats (to the security of a computing system), the security objectives for the plug-in aims to **prevent** [3]:

1. Interruption - an Application that uses the plug-in shall not be made unavailable or unusable, even if an error occurs
2. Interception – an Application that uses the plug-in shall not be forced to have "open" classes (too much API possibilities to extract the class information).
3. Modification - the plug-in shall restrict and force a specific behavior of the template classes that are intended to be used or extended by domain specific classes.
4. Fabrication - the plug-in must guarantee that Report objects and template classes being created are valid and conform to the structure that is imposed of the framework (Pfleeger do not categorize software to have fabrication threats, but in this paper it seemed applicable).

4. Setting the scope of the security analysis

Java relies on software technology to ensure security, each program run within a JVM[11]. This concept is often referred to as the sandbox model, where the JVM in combination with the following parts ensure that the sandbox works correctly [4]; Security manager - checks and restricts access in advance; Compiler and a byte code verifier - ensures that only legitimate java code is executed; Classloader - defines a local namespace to avoid interference between programs.

Also, the Java language has built-in security features such as automatic memory management, garbage collection, type-safety (for both compile time and runtime) and range checking on strings and arrays. This concept is a well known and proven technology, however the environment does not ensure that the executed Java code is secure. It is more likely that the security flaws within a computing system relate to the program itself (than the environment) and as defined in the journal "A Taxonomy of Computer Program Security Flaws" [5], a security flaw is defined as "part of a program that can cause the system to violate its security requirements". This paper does not examine security in the Java environment, but looks for security flaws in the code to understand what is needed to "meet the security requirements in the first place instead of attempting to mend the flawed systems already installed" [5].

To be able to assess the security of a computing system, an analyst must understand the system thoroughly and (based on this knowledge it is possible to) find its vulnerabilities [5]. Section 2 explained the functionality of the application and this section sets the scope for what this paper will look for. The code to analyze is categorized according to the taxonomy by the following vulnerabilities:

C) Security flaws by how they occurred, **introduced inadvertently:**

1. Validation error:
    a. Failing to check the type of parameters supplied or if checks are misplaced.
    b. Access permissions of a file.
    c. Different interface routines to a data structure fail to apply same set of checks.
2. Domain error:
    a. Object reuse or residuals from a previous object instance.
    b. Invalid object state.

[11] JVM = Java Virtual Machine, each java program runs in an isolated environment, where all access with the outside world is checked according to defined access permission policies.

3. Serialization:
    a. Parameters changed after a certain checkpoint where not allowed.
    b. Different system components causing security violation.
4. Boundary condition violation:
    a. Omission of constraint checks.
    b. Unnecessary resource consumption.
5. Logic error, code bugs.

D) Security flaws by time of introduction, **during development,** in source code:

1. Inadequately defined (or named) module, interface, method or variable (class member).
2. Misunderstanding of the meaning of parameters.
3. Introduced mechanisms for debugging and testing.

5. Gathering common software security guidelines

To learn how to write secure code from scratch this section collects and evaluates common software security guidelines, especially by looking at general programming principles, previous security audits on related work and by following expert opinions within the security field.

5.1 General programming principles

Pfleeger emphasize that "an inadvertent error can cause just as much harm…as can an intentionally induced flaw" [3]. This is an important observation and by following general programming principles one can reduce code mistakes, which should be useful to investigate in this security audit.

E) Important questions based on general programming principles are:

1. How well is modularity achieved?
    - Make classes small and easy to understand.
2. Do the classes provide sufficient encapsulation?
    - Limit the effect one module can have on another.
3. Do the public classes have the appropriate level of information hiding?
    - Prevent manipulation by hiding how a module works.

How do these questions fit into the security perspective? With modularity or "economy of mechanism", as Saltzer and Schroeder [7] asserted already 1975, it is better to have smaller classes where the design becomes simpler and easier to inspect and trust. Also, if a class have many access paths and dependencies, it will more likely bypass an inspector's eye, because of the hiding effect when "unwanted access paths will not be noticed during normal use" [7]. Loosely coupled modules should improve the likelihood of implementation correctness and also benefits such as: easier to maintain, understand, reuse, correct and test [3]. Modularity accelerates other advantages like encapsulation and information hiding (some security benefits are captured in list E above [3]).

5.2 Previous security audits on related work

This section attempts to improve program robustness by examining previous identified security flaws on similar technologies. In the first release of Java the researchers at Princeton University found several security flaws that forced Java to be redesigned [3].

F) The list below is an excerpt from the security flaws identified (the nested level is an idea of how this can be correlated to Xerpt):

1. Absence of a well-defined security policy.
    - This paper defines one now for Xerpt.
2. Lack of security mechanisms that is always invoked.
    - Need to check that default state is always set and central Report generation mechanisms are always achieved for the different output formats.
3. Lack of a tamperproof security mechanism.
    - Is it possible to tamper the inner parts of the report object once it is set up?
4. Lack of a security mechanism that is small and simple.
    - Is the report generation mechanism well separated, is file management well separated?
5. Lack of a trusted computing base.
    - Is the code well structured, are unnecessary external libraries used?
6. Lack of modularity and limited scope.
    - Shall the scope of the report object be restricted? Is class composition too complex?
7. Lack of defense in depth.
    - What happen if the Report generation fails at some level?
8. Lack of logging and auditing.
    - Does Xerpt, as a plug-in, need logging and auditing in the aspect of finding errors?

G) Design principles for protection mechanisms.
Saltzer and Schroeder, 1975 [7].

1. Principle of Economy of Mechanism.
   - Simple and small design.
2. Principle of Fail-safe Defaults.
   - Deny access by default.
3. Principle of Complete Mediation.
   - Check every access to every object.
4. Principle of Open Design.
   - Design should not be secret.
5. Principle of Separation of Privilege.
   - Have several keys for the protection mechanism (must be put together).
6. Principle of Least Privilege.
   - Force every process to operate with the minimum privileges needed to perform its task as short time as possible.
7. Principle of Least Common Mechanism.
   - Share as little as possible among users.
8. Principle of Psychological Acceptability.
   E

H) Guiding principles for software security.
Viega and McGraw, 2001 [2].

1. Secure the weakest link.
   - Find the weakest parts of the system.
2. Practice defense in depth.
   - Have several defenses against errors.
3. Fail securely.
   - Plan for failures that are unavoidable.
4. Follow the principle of least privilege.
   - See G-5 to the left.
5. Compartmentalize.
   - Separate and isolate a system in small units to minimize the damage.
6. Keep it simple.
   - Avoid complexity = avoid problems.
7. Promote privacy.
   - Try hard to keep user data private.
8. Remember that hiding secrets is hard.
   - Keep secrets outside the code.
9. Be reluctant to trust.
   - Be skeptical to everything.
10. Use your community resources.
    - More eyeballs = more validation.

Figure 5. Conceptual principles for implementing secure software.

I) Fundamental principles of software engineering.
Pfleeger, 1997 [3]

1. Modularity - Write code in small self-contained “modules”.
   a. Divide tasks into subtasks and let them perform separate parts of the task.
   b. Make each module to have only one purpose and easy to understand.
   c. Strive to get loosely coupled modules.
2. Encapsulation - Isolate the negative effects between modules.
   a. Minimize coupling between objects.
   b. Prevent unwanted access from the outside or change of behavior.
   c. Have well-defined interfaces.
   d. Minimize the shared, modifiable information between modules.
3. Information hiding - hide how a module performs its task.
   a. Conceal the way a module does its task; make a “black box”.
   b. Limit the effect modules have of each other.

J) Twelve rules for developing more secure java code
McGraw and Felten, 1998 [1]

1. Don't depend on initialization.
2. Limit access to your classes, methods, and variables.
3. Make everything final (unless there's a good reason not to).
4. Don't depend on package scope.
5. Don't use inner classes.
6. Avoid signing your code.
7. If you must sign your code, put it all in one archive file.
8. Make your classes noncloneable.
9. Make your classes nonserializeable.
10. Make your classes nondeserializeable.
11. Don't compare classes by name.
12. Secrets stored in your code won't protect you

Figure 6. Concrete rules for how to implement secure software.

5.3 Expert opinions within the security field

The Figure 5 and 6 summarize expert opinions in order to provide guidance from the conceptual level down to code level. Notice that the conceptual principles within software security have many similarities and have not changed much over a time period of more than 25 years. Each criterion is considered and the ones that apply for this case study are identified.

Analysis of G: Principle 1-3, 6-8 shall be investigated. Make classes small and simple, deny access by default because “a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use”, check every access to every object, minimize privileges (prevent accidental mistakes, e.g. overwrite an existing file without asking), share as little data as possible between users (e.g. shared variables and also audit common code more carefully) and make the system easy to use (should not place too much burden on its user with a complex design) [7]. Principle 4 and 5 are not applicable or omitted for this audit.

Analysis of H: Principle 1-7 shall be investigated. Find the weakest parts of the system, have several layers of defense, fail the system correctly and securely if it is unavoidable (if wrong arguments are supplied then exit the system and print out an error), minimize the privileges to perform an operation and limit its time granted, isolate parts of a system from the effects of other parts, avoid problems by avoiding complexity, with simplicity comes usability and keep information private. Principle 8, 9 and 10 are not applicable or omitted for this audit.

Analysis of I: These are fundamental programming principles and will be carefully analyzed.

Analysis of J: Rule 1-3, 5, 8 and 11 shall be investigated. Strive to make all classes, methods and variables private. Use final to specify the intention, otherwise the class can be extended in a dangerous or unforeseen way [1]. Rule 4, 6, 7, 9, 10 and 12 are not applicable or omitted for this audit.

6. Performing the security audit

Two developers with certification in Java knowledge according to the “Sun Certified Java Developer” degree tested Xerpt using this document as test specification and the checklist (see the Appendix in the end) to record found flaws.

The audit process was improved continuously and the changes are documented below. The first code analysis took two hours for one small class (`XMLTransformer`), which was way too long time. All the selected rules and principles were analyzed and many of them seemed to be overlapping and also affect different aspects of the system. The initial checklist had all titles for labeled lists (A-J) in sequence. Also, it was hard to understand what to look for, when reading each principle and rule. The checklist would have been easier to follow if everything to look for was listed on it. The analyzed classes that did not have any errors were often forgotten to be recorded on the checklist.

Improvements: Explanations were provided on the principles and rules in the document. The labeled lists were grouped on the checklist in the following categories:

- Boundary – analyze the boundaries of Xerpt (see Figure 7 below) versus an Application, versus input sources and output destinations.
- Package – analyze the classes from the package perspective.
- Class – analyze the classes at the lowest detail.

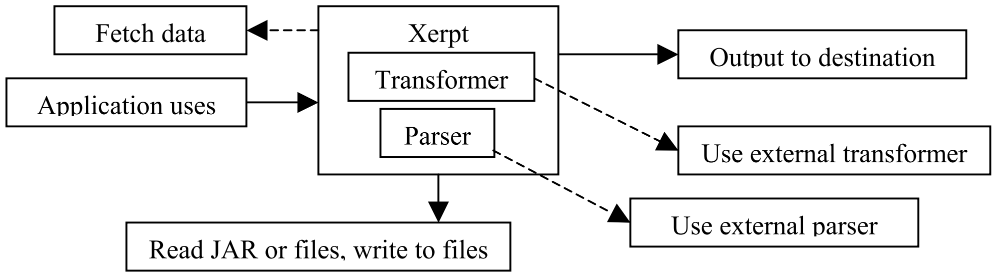


Figure 7. The boundaries for Xerpt.

A constraint was added: At least one category was required to be completed for a checklist form to be complete. Also, instructions were added to the checklist e.g. to be able to specify what was analyzed, improvements suggestions (solutions), how to fill in the matrix etcetera. A column far to the right was added to specify what was tested (to avoid to forget to report the ones that were OK).

7. Identified program flaws

The classes for Report generation were selected for the security audit and the labeled lists were used for testing. The lists surprisingly seemed to work well for any perspective (boundary, package and class). Table 1 shows high average detected flaws for lists A, B, C, G and J. G & H and E & I checked equivalent things, therefore only one of them needed to be analyzed to get appropriate coverage.

List	A	B	C	D	(E)	F	G	(H)	I	J
Flaw count	37	24	82	12	13	18	17	1	9	84
Used times	3	2	4	3	3	3	1	1	2	4
Average	**12.3**	**12**	**20.5**	**4**	**4.3**	**6**	**17**	**1**	**4.5**	**21**

Table 1. Average flaws per list.

An average of 11.4 flaws per list was detected, but many of them came from J3 (60 times, omitted "final" on classes, methods and variables) and C1a (40 times, omitted checks on parameters). Without counting them an average of 7,5 flaws per list was detected. Important to mention is that the lists do not have equal amount of “questions”, they look at different aspects of security and do not distinguish between serious or harmless flaws.

Class tested	Lists used	Serious flaws	Other Flaws	Main weaknesses
`AbstractReport`	A,B, G,E, F	10	36	Not protected scope, throw different types of errors, validate return values from implemented methods, need to check DTD and XSL file paths and extends an inner class from another class.
`Report`	C,J	3	46	Constructor not safe, everything shall be final, separate the inner class and remove dead code.
`ReportManager`	C,D, I, J	4	45	Tied too much to HTMLViewer, which does transformation (should not), does not take care of error from classloader, separate loadJARContent mechanisms, not package scope.
`ReportPart`	A,B, E, F	3	47	Is not package scope, type constants easy to misuse, tag names unchecked, unsafe constructor, make everything final and check parameters.
`XMLParser`	C,D, J	0	28	Remove dead code, fails to check parameters, resource consumption, debug mechanisms, not final, not package scope.
`XMLTransformer`	A,C,D, E, F, H, I, J	4	70	Not final, do not verify DOM against DTD, redundant code, not package scope, debug methods from not trusted external library.
	26	**24**	**272**	

Table 2. Flaws per class with main weaknesses.

As depicted in Table 2, code visibility (public, protected, package, private) was a frequent problem, the constants for the parameters did not protect against invalid values, insufficient file handling, no XML validation and unsafe handling of constructors were other serious flaws. Also, the code could make use of another refactoring to remove redundancies, get rid of debug code and remove “dead” (commented out) code.

8. Discussion

Generally, it was hard and time consuming to analyze each class to address security, because it requires that you need to know what you are looking for. All lists of principles and rules catches flaws quite well, especially C and J for low level things. The security objectives, which also were effective to test the security with, were not fulfilled in A1 (spare the Application from the complexity), A5 (unintentional or erroneous usage), B3 (restrict and force specific behavior) and B4 (guarantee that only valid objects can be created that follows the required structure). The lists were grouped to specify boundary, package and class perspective, which reduced the overlapping between security principles and rules.

Decision about improvements: The obvious needed improvements are to; add checks on almost all parameters; add “final” almost everywhere; reduce visibility on classes, methods and variables; restrict and improve constructor creation; constants for parameters need to be changed to something more type-safe (see Figure 4, class `Report` and `ReportPart`); throw more specific exceptions for different errors; file handling and XML handling could be done more correctly.

Security vs. architecture: The inner class in `Report` increased the complexity of the class and the subclasses, flaws were harder to find, especially when modeling tools do not visualize inner classes in diagrams. In several classes many problems could have been reduced if the code was normalized as much as possible and forced to have only one purpose and divide tasks in subtasks. Security would gain architecture and vice versa.

Effects on framework design: Abstract and complex classes need to be provided with examples. Also, all generic or framework centric code (like the template classes) must be designed as robust as possible.

Reusable parts of this project: The Figure 5 and 6 are gathered on one page that can help to write more robust and secure code. Also the labeled lists state useful security opinions and in combination with the checklist (in the Appendix, to record identified flaws) the concept could easily be adopted for another Java application. Also, using Pfleeger's security goals and threat-definition (see list A and B) and the classification from the program flaw taxonomy (see list C and D) assisted well in forming the security objectives and to find flaws.

9. References

[1] G. McGraw, E. Felten, "Twelve Rules for Developing More Secure Java Code", JavaWorld, 01 Dec. 1998, http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules_p.html, June 2002.

[2] J. Viega, G. McGraw, "Building Secure Software", Addison-Wesley, 2001.

[3] C. Pfleeger, "Security in Computing", 2/e, Prentice Hall PTR, 1997.

[4] L. Gong, "Secure Java class loading", IEEE Internet Computing, vol.2, no.6, Nov.-Dec. 1998.

[5] Landwehr CE, Bull AR, McDermott JP, Choi WS. "A taxonomy of computer program security flaws", [Journal Paper] ACM Computing Surveys, vol.26, no.3, pp.211-54. USA, September 1994.

[6] XML parsing and transformation, see http://xml.apache.org/xalan-j/index.html and especially http://xml.apache.org/xalan-j/design/design2_0_0.html about Xalan 2 design, June 2002.

[7] J. H. Saltzer, M. D. Schroeder, "The protection of information in computer systems", Proceedings of the IEEE 63, 9 September 1975, http://www.cs.virginia.edu/~evans/cs551/saltzer/, June 2002.

	Checklist – Code security flaw observations Look up the labeled lists (e.g. A, B, C) in the other sections of this document to understand what to investigate on each statement below. **Date:** **Analyzed package or class:** Complete at least one of the categories (boundary, package, class).	Enter the analyzed ID (e.g. A2, C1b) in the cell matching approx. number of flaws.									Enter tested ID range, e.g. C1-5.
		Many flaws				**Few flaws**					
		25	**20**	**15**	**10**	**5**	**4**	**3**	**2**	**1**	
Boundary perspective	**B) Security objectives, failure to prevent threats:**										
	G or H) Flaws identified by experts' conceptual principles:										
Package perspective	**A) Security objectives, flaws in functionality provided:**										
	E or I) Neglecting good programming principles:										
	F) Occurrence of flaws similar to related work:										
Class perspective	**C) Scope, vulnerabilities introduced inadvertently:**										
	D) Scope, vulnerabilities inserted during development:										
	J) Flaws identified by experts' concrete rules:										

Overall, what were the code's weaknesses and how would you improve the code?