TOWARDS SYSTEMATIC SOFTWARE SECURITY HARDENING

MARC-ANDRÉ LAVERDIÈRE-PAPINEAU

A THESIS IN THE
CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE (INFORMATION SYSTEMS SECURITY)
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

JULY 2007

© MARC-ANDRÉ LAVERDIÈRE-PAPINEAU, 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-34444-6
Our file Notre référence
ISBN: 978-0-494-34444-6

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Towards Systematic Software Security Hardening

Marc-André Laverdière-Papineau

In this thesis, we report our research on systematic security hardening. We see how the software development industry is currently relying on highly-qualified security experts in order to manually improve existing software, which is a costly and error-prone approach. In response to this situation, we propose an approach that enables systematic security hardening by non-experts. We first study the existing methods used to remedy software vulnerabilities and use this information to determine a classification and definition for security hardening. We then see how the state of the art in secure coding, patterns and aspect-oriented programming (AOP) can be leveraged to enable systematic software security improvements, independently from the users' security expertise. We also present improvements on AOP that are necessary in order for this approach to be realizable. The first improvement, *GAFlow* and *GDFlow*, two new pointcut constructors, allow the injection of code that precedes or follows any of the points in the input set, facilitating the development of reusable patterns. The second, *ExportParameter* and *ImportParameter*, allow us to safely pass parameters between different parts of the program. Afterwards, we leverage our previous findings in the definition of *SHL*, the Security Hardening Language. *SHL* is designed in order to permit language-independent expression of security hardening plans and security hardening patterns in an aspect-oriented manner which enables refinement of patterns into concrete solutions. We then demonstrate the viability of this approach by applying it to add a security feature to the APT package acquisition and management system.

Acknowledgments

Whatever you do in word or deed, do all in the name of the Lord Jesus, giving thanks through Him to God the Father. (Colossians 3:17)

Putting this in practice, I first thank my beloved God, who gave me the opportunity to start this degree, His constant love to persevere through, and His peace to complete this thesis. In short, thanks to Him,

I can do all things through Him who strengthens me. (Phillipians 4:13)

I will also thank my parents for their unquestionable support for my studies, and for all the patient years they spent helping me get where I am at now.

I would like to thank my supervisor, Prof. Dr. M. Debbabi, for providing me an opportunity to accomplish this visionary, exciting and challenging research. This work would not have been possible without his guidance, suggestions, and critical remarks.

I must also thank the industrial partners behind the Trusted Free Open Source Software (TFOSS) project for their financial and in-kind contribution to the project.

Also, a very special thanks to all my TFOSS team colleagues and the members of the Computer Security Laboratory for all their collaboration in this research and making the lab an enjoyable work environment, and most notably to Azzam Mourad, with whom I conducted this research in very close collaboration. I also thank my friend and classmate Serguei Mokhov for his hardworking spirit, great technological skills, encouragement and friendship during and after the courses that we had to take together. He, with Djamel Benredjem's help, supported the study of the Linux kernel.

I would also like to thank Thomas Würthinger for providing me access to his Hotspot Client Compiler Visualizer. My understanding of control flow graphs would be more limited without it.

Finally, my gratefulness extends to friends who contributed their time to help me review this Thesis, most notably Ann Fry and Rebecca Goldman.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms and Definitions

API	Application Programming Interface
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
CFG	Control Flow Graph
Cflow	Control Flow pointcut in AOP languages
CGA	Closest Guaranteed Ancestor
CIA	Confidentiality, Integrity, Availability
COTS	Commercial Off The Shelf Software
FOSS	Free and Open Source Software
GAFlow	Guaranteed Ancestor Control Flow
GDFlow	Guaranteed Descendant Control Flow
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol over SSL
Pcflow	Predictive Control Flow pointcut
SSL	Secure Socket Layer protocol
TLS	Transport Layer Security, the successor of SSL

Advice	In AOP, code to be injected at locations specified by a pointcut
GnuTLS	Open source library for the TLS protocol
Pointcut	In AOP, an expression selecting program locations for code injection
Weaver	AOP tool integrating advices to the base code

Chapter 1

Introduction

Computer security is taking an increasingly predominant role in today's environment. The information technology industry is facing challenges in public confidence at the discovery of vulnerabilities, and customers are expecting security to be delivered out of the box, even on programs that were not designed with security in mind. Software maintainers must now face the challenge to improve programs' security and are often under-equipped to do so. In some cases, little can be done to improve the situation, especially for Commercial-Off-The-Shelf (COTS) software products that are no longer supported, or for programs for which the source code is lost. However, whenever the source code is available, as it is the case for Free and Open Source Software (FOSS), a wide range of security improvements are possible.

Computer security professionals have been promoting tools, best practices, and guidelines to be used by the software development industry, with little adoption so far. Developers, often pressed by a dominating time-to-market priority, must deal with a large set of technical and non-technical issues, in which case security concerns are not thoroughly

addressed. The practical help for developers are typically centered around frameworks, standards and design guidelines, which can be of limited use for both implementers and maintainers.

In this thesis, we present research results of an approach designed to facilitate security hardening by systematically injecting security hardening code at required program locations, and early results from the TFOSS project.

1.1 The TFOSS Project

The Trusted Free and Open Source Software (TFOSS) project is a research initiative lead by the Computer Security Laboratory in the Concordia Institute for Information Systems Engineering, at Concordia University. It is supported by a NSERC-DND grant, in collaboration with the following industrial partners:

- Defence Research and Development Canada, Valcartier Research Centre (DRDC Valcartier), notably with the help of Robert Charpentier and Frédéric Michaud
- Bell Canada, notably with the help of Alan Bernardi, Claire De Grasse and Glenn Henderson

The project aims at the creation of bleeding edge tools for vulnerability detection and security hardening of open source software. The following sub-projects are currently being researched:

Static Detection of Vulnerabilities: Enhancing the state of the art in static analysis tools, this team develops new tools to detect low-level security vulnerabilities.

Dynamic Detection of Vulnerabilities: This team greatly innovates by developing tools that perform code instrumentation that detects and reports vulnerabilities at runtime.

Formal AspectJ Weaving Semantics: This team is formalizing the weaving operation of the AspectJ tool, which will enable formally proving the effectiveness of security hardening.

Implementation of Security-Related Aspect-Oriented Primitives: This team is implementing security-related pointcuts and other primitives, that were the subject of theoretical proposals only.

Security Hardening and Patterns: This team aims at creating new methods to leverage and enrich aspect-oriented technologies and patterns in order to enable systematic, and eventually automatic, security hardening of software at both the code and design levels.

The author is part of the last team, and has contributed to both theoretical and practical aspects of its research, with a distinct focus on the experimental aspects.

1.2 Problem Statement

Software security improvements are currently performed manually, by security experts, which has been demonstrated as an error-prone, time-consuming and expensive process. For instance, a security push organized for the Windows 2003 Server lasted for more than a month, and still saw major security issues appear at the last minute [31, Ch. 3]. Since the software industry is demanding high security software, and the current approach is showing itself to be flawed in both practical and economic grounds, it becomes necessary to determine how existing software can be improved without such overhead. Furthermore, since it is known that software is able to deal with complexity in a better way than humans

tend to, it becomes worthwhile to examine an approach that systematically applies security improvements over software as a first step towards automatic security hardening.

The current practices in the industrial and research communities do not answer well to this need. The most prominent proposals could be classified into security design patterns, secure coding and security code injection using aspect-oriented programming (AOP). We will investigate them carefully in Chapter 2, but briefly introduce them here.

Many Security Design Patterns are available in order to guide software engineers in securely designing their applications. They offer proven solutions to commonly occurring design problems [11]. These patterns are however of a lower relevance for security hardening, as our task typically fits in the maintenance part of the software lifecycle. The secure coding literature, [7, 9, 30, 62, 73] presents safe programming techniques, typically as list of programming errors together with their corresponding solutions, and often focus on the `C` and `C++`programming languages. Their intent is to instruct software developers to avoid these errors. Although this information is highly pertinent, it is foremost a means to educate developers into becoming security experts using a manual approach to vulnerability remedial, thus perpetuating the current state of affairs.

Aspect-Oriented Programming (AOP), is a programming paradigm based on the idea that computer systems are better programmed by specifying the various concerns separately from the core program logic, and then relying on an underlying infrastructure to compose them together. The techniques in this paradigm were precisely introduced to address the development problems that are inherent to crosscutting concerns, and the usefulness of this approach for the injection of security code has been demonstrated. However, the available contributions [12, 20, 32, 63, 66] only propose a simpler approach for security experts to

inject the required code, but do not necessarily empower maintainers to do so.

In short, we see that security design patterns, secure programming books and AOP technologies all offer a piece of the puzzle, but fail to offer a complete solution that enable non-experts to systematically implement security improvements on software.

1.3 Objectives

This research aims at building the theoretical foundation for systematic security hardening. In order to do so, we must first study software vulnerabilities that are encountered in software. With this knowledge, we must then compile and organize the solutions existing to these vulnerabilities into a useful taxonomy. After, we must assess whether or not AOP technologies are fit to answer our needs. If these technologies are indeed appropriate, we must determine which improvements to AOP languages would facilitate security hardening activities. The next objective is the elaboration of an approach that adapts AOP to the field of security patterns in order to allow the specification of security hardening patterns. The approach must also include security hardening plans that specify what security improvements must be implemented where. This strategy will enable the dissociation of the decision-making stage from the implementation. In order to facilitate this specification, we must also create a specialized language for the specification of plans and patterns.

The direct consequence of those objectives is the enabling of non-experts to perform the implementation part, and allowing experts to focus on the task of choosing the best option available. It will also lower the level of digging necessary to harden an application, lower the time required in order to make the necessary modifications, and enable reuse.

1.4 Contributions

In this thesis, we contributed to the state of the art in security hardening in the following ways:

Definitions and classifications: After studying the practices in the industry, notably in free and open source software, we defined security hardening as well as offered classifications of their practices.

Applicability of AOP for security hardening: By reviewing the literature and experimenting with the technology, we established that AOP was appropriate for the purpose of security hardening, despite shortcomings we identified.

Innovative AOP constructs: We proposed four AOP constructs, *GAFlow* and *GDFlow*, as well as *ExportParameter* and *ImportParameter*, in order to enrich current AOP languages and enable the specification of security hardening patterns.

Creation of an approach for systematic security hardening: We described how users could use a combination of security hardening plans and security hardening patterns in order to apply security improvements in a systematic manner.

Elaboration of SHL*:* We designed *SHL*, a language dedicated for the specification of hardening plans and patterns.

Experimental Validation: We performed a case study in order to validate the applicability of our approach.

1.5 Thesis Organization

This thesis is organized as follows. In Chapter 2, we introduce the reader to the foundations of computer security and aspect-oriented programming, where we also present the state of the art in research that was foundational to this work. Then, in Chapter 3, we present a classification of software security hardening practices, and then show how the practices used in the industry fit within this classification. Chapter 3 also shows code-level hardening practices in the Linux kernel development team. Afterwards, in Chapter 4, we will see a novel aspect-oriented approach to the problem of security hardening. This approach uses *SHL* to write language-independent security hardening patterns and plans. Chapter 5 is built as a logical extension of the previous one, where we will see how the experiments done in early validation of our approach demonstrated shortcoming in AOP technologies, and what solutions we found to these issues. Experimental results of the application of our approach on APT, a package acquisition and management system for the Debian platform are then shown in 6. We finally bring concluding remarks in Chapter 7.

Chapter 2

Software Security

We are interested in the improvement of software security for already existing software at the implementation and design levels. This choice was motivated by the need to improve the methods available to software maintainers in the industry. Software security is mostly interested in maintaining the extended Confidentiality-Integrity-Availability (CIA) properties of applications [69], which are defined in [65] as:

Confidentiality: "The property that information is not made available or disclosed to unauthorized individuals, entities, or processes [i.e., to any unauthorized system entity]."

Integrity: "The property that information has not been modified or destroyed in an unauthorized manner."

Availability: "The property of being accessible and usable upon demand by an authorized entity."

Accountability: "The property of a system (including all of its system resources) that ensures that the actions of a system entity may be traced uniquely to that entity, which can be

held responsible for its actions."

Assurance: " An attribute of an information system that provides grounds for having confidence that the system operates such that the system security policy is enforced."

In the computer security milieu, a vulnerability is defined in [65], as "A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy." For the sake of simplicity, we will henceforth use a simplified division of security vulnerabilities among the lines of implementation ("low-level") and design ("high-level").

The rest of the chapter is divided as follows. In Section 2.1, we will cover the topic of secure programming. Afterwards, in Section 2.2, we will study the design of secure systems. In Section 2.3, we see how Aspect-Oriented Programming can be used to secure applications. Finally, in Section 2.4, we summarize what we covered here.

2.1 Secure Programming

The field of secure programming focuses on avoiding common programming mistakes that result in security vulnerabilities. As such, this field is highly tied with the technology used, typically being the programming language and the operating system, while also encompassing the proper use of cryptography.

A lot of research has been published on the field of secure programming for `C` and `C++` programs, since a large amount of production-level software has been written in those languages. The design of those languages also allows the existence of many types of security vulnerabilities that are not possible in modern languages. These older languages require

manual memory management and and offer low type safety, both being sources of many programming errors. Furthermore, since these languages are capable of interacting with the operating system directly, they are able to use system resources in a manner that is not always safe. It is thus very important to understand the security implications of improper C and C++ programming and to learn good programming practices to avoid them.

We will now proceed to a non-exhaustive survey of existing platform-independent vulnerabilities, before examining the available contributions in the field.

2.1.1 Implementation-Level Vulnerabilities

Implementation-level (also known as low-level) vulnerabilities are associated with programming languages as well system/library features. In the example of C programs, many errors are related to the lack of bound checking in the language itself as well as for manual memory management. We will now look at buffer overflows, integer vulnerabilities and memory management issues. Please note that many other categories of security issues are present in C programs, and that we only present here an introduction to the topic.

Buffer Overflows

Although buffer overflows are typically the results of improper bound checking, flawed integer logic, and missing input validation, they are nevertheless in a category of their own due to their widespread nature and severity. Buffers on both the stack and the heap can be corrupted, results of which can range from denial of service up to remote control of the computer [30].

Integer-Related Vulnerabilities

Integer security issues arise on the conversion (either implicit or explicit) of integers from one type to another, and because of their inherently limited range [62]. C compilers distinguish between signed and unsigned integer types and silently perform operations such as implicit casting, integer promotion, integer truncation, overflows and underflows.

Such silent operations are typically overlooked, which can cause various security vulnerabilities. These may be used to write to an unauthorized area of memory, causing corruption or denial of service. They can also cause other security problems by bypassing preconditions and expected protocol values that are specific to the program exploited. The following are the common causes of these vulnerabilities:

Integer Sign Conversion: Because the C language conserves the bit pattern when converting between signed and unsigned integers of the same size, we can find negative or unexpectedly large values when not expected. This is due to the fact that most processors store signed integers with the first bit as the sign bit, whereas the same bit is used by unsigned integers in the same way as other bits.

Integer Signedness Errors: Signedness errors occur when the program expects an unsigned value, but instead finds a signed one. Because of the inappropriate assumption, the program does not validate if the value is positive, potentially resulting in a security vulnerability. These errors typically happen in conjunction with conversion errors.

Integer Truncation Errors: A truncation error occurs when an integer is converted to one of a smaller type. The bit pattern of a subset of the original integer is preserved as is. If the smaller type is signed, this may result in a negative value.

Overflow and Underflow: Integer overflow and underflow happen by the following means: Adding or multiplying beyond the integer's maximum value, dividing by -1 (overflow) or subtracting below its minimal value (underflow). This will result in errors similar to integer conversion. It is also noteworthy to remember that unsigned integers obey modular arithmetic rules in case of overflow, resulting in smaller values than expected, but that are still positive.

Memory Management Vulnerabilities

The `C` programming language allows programmers to dynamically allocate memory for objects during program execution. `C` memory management is an enormous source of safety and security problems. The programmer is responsible for pointer management, buffer dimensions, allocation and deallocation of dynamic memory space. Thus, memory management functions must be used with precaution in order to prevent memory corruption, unauthorized access to memory space, buffer overflows, etc. The following are the major errors caused by improper memory management in `C`.

Using Uninitialized Memory: In `C` programming, the memory space pointed to by newly declared pointer is not initialized, and it typically points at a random location. The consequence of dereferencing these pointers include denial of service, information disclosure, and memory corruption.

Accessing Freed Memory: A pointer on which the `free` function was called can still be accessed. It is however possible that the memory range was allocated for another use, and that this access will result in the reading of invalid data or the corruption of data used in

another part of the process.

Freeing Unallocated Memory: The `free` function must be called on a memory location previously allocated using the `alloc` family of functions. Otherwise, freeing an unallocated memory location can cause memory corruption and denial of service. This problem arises because the `free` call is performed with the pointer uninitialized, either through programming error or a failed memory allocation.

Memory Leaks: The dynamically allocated memory must be freed after usage and before the pointer to its location goes out of scope. Failure to do so results in a memory leak. Memory leaks degrade performance and can cause a denial of service.

2.1.2 Secure Programming Literature for `C` and `C++`

Secure programming books often introduce the reader to the world of software vulnerabilities in `C` and `C++` and their solutions. This literature will often list various forms of software vulnerabilities, show code examples that have vulnerabilities, and explain them. They will also often propose a method that can be used to remediate the vulnerability. As such, these proposals were a cornerstone for this research. We mention here the most important references we used in our research, although the field of secure programming is far from lacking authors and references to mention here.

One of the newest and most useful additions is from Seacord [62], who offers in-depth explanations on the nature of all known low-level security vulnerabilities in `C` and `C++`, and covers integer-related vulnerabilities in more depth than other sources.

Another common reference is from Howard and LeBlanc [30], and includes all the basic

security problems and solutions, as well as code fragments of functions allowing to safely implement certain operations (such as safe memory wiping). The authors also describe high-level security issues, threat modeling, access control, etc.

Bishop, via slides and a book [7, 9], provides a comprehensive view on information assurance, as well as security vulnerabilities in `C`. In addition, these contributions provide some hints and practices to solve some existing security issues for UNIX `setuid` programs.

Wheeler [73] offers the widest-reaching book on system security available on the web. It covers operating system security, safe temporary files, cryptography, multiple operating platforms, spam, etc. Furthermore, it includes a very thorough treatment of the problem of insecure temporary files.

The Secure Programming Cookbook for C and C++ [72] is a hands-on solution for programmers looking for direct solutions to typically-encountered security problems. The authors mention recipes for safe initialization, access control, input validation, cryptography, networking, etc.

2.1.3 Evaluation

The field of secure programming offers many highly relevant, although sometimes repetitive, contributions that can drastically help programmers write secure code. They are also of help for maintainers who need to improve the security of existing systems. Its only limitation is that the approach aims at educating programmers in order to help them make better human decisions. But since the approach is not systematic, it cannot prevent human errors

from being made. As such, the secure programming references should be used as a primary resource for the construction of security hardening patterns dealing with lower-level security issues.

2.2 Secure Software Design

As important as secure programming is to software security, it remains that no amount of attention to detail in the implementation phase can compensate for poor system design. Many access control approaches have emerged in this area, with the UNIX, Bell-LaPadula and Chinese Wall models being of particular theoretical interest [9]. However, secure application design goes further than the design of an access control system, but constitutes a specific branch in the field of software design. This sub-field integrates the concepts of security components, communication mechanisms and protocols, operating system dependent design impacts (especially for access rights and privileges), etc.

Bishop describes the following secure software design principles [9]:

Least Privilege: Only the privileges needed to perform a given task should be granted to the principal performing the task.

Fail-Safe Defaults: The default access rights to an object should be "denied".

Economy of Mechanism: Security mechanisms should be as simple as possible.

Complete Mediation: All accesses to an object must be validated before being granted.

Open Design: The security of a mechanism should not rely on the secrecy of its design or implementation.

Separation of Privilege: Security decisions must rely on multiple conditions to be satisfied.

Least Common Mechanism: Mechanisms to access resources should not be shared.

Psychological Acceptability: The presence of a security mechanism should not make the resource more complicated to access for authorized principals.

Beyond these principles, some authors have also suggested software design advice. Bishop, in a set of instructional slides, offers advice on dividing software in processes that have exactly the right level of privilege to perform their task [8]. In [9], the reader will also learn secure software design from design principles and case studies. Howard and LeBlanc [30] suggest the use of threat modeling as a tool for correctly choosing the right security mechanisms and their proper deployment in systems. It shows how secure software should not be structured in an arbitrary manner, but that design decisions should be directly correlated with the requirements and policies. In [31], Howard and Lipner describe the process used by Microsoft to create secure software, named the Security Development Lifecycle. They show a methodology used in this approach for design named attack surface reduction. Attack surface reduction is a strategy that dictates the reduction of entry points, of the privileges and the amount of executing code. One particular concern it deals with is the presence of anonymous paths that are of a higher risk. Graff and Wyk [24] have also written on the principles and practices behind the construction of secure software. Their book covers all the steps of the software development process, and includes a coverage of security architecture and design.

All these books are useful, but often lack very direct advice on how to design applications well. Security patterns have emerged in order to answer this need.

2.2.1 Security Patterns

The idea of patterns was introduced by Alexander et al. [3] in the field of architecture and urbanism, and was later reused in the object-oriented world. Patterns are proven solutions to commonly-occurring problems. A pattern is commonly structured with a summary, a description of the problem it tries to solve, a description of the solution and a graphical representation of its structure. Security patterns are similar patterns, but applied for information security.

Security design patterns encapsulate expert knowledge in the form of proven solutions to common security problems [60]. These patterns will fit at different levels of abstraction and areas of concerns, resulting in many patterns that can be different from the design patterns the reader may have been used to. There has been an increasing number of publications in this field, and the following shows the major contributions that were available when this research was done, in mid-2005, with minor updates.

Yoder and Barcalow, in [75], introduce a 7-pattern catalog that could be combined together as the design of a secure system. It is interesting to note that these patterns emerged from experimental results adding security in the Caterpillar/NCSA Financial Model Framework.

Kienzle et al. [36, 37] created a 29-pattern security pattern repository for web applications. These patterns were categorized as either structural or procedural. Structural patterns are implementable patterns in an application whereas procedural patterns are patterns that were aimed to improve the development process of security-critical software.

Romanosky [57] introduces another set of security patterns. The discussion however has focused on architectural and procedural guidelines more than security patterns.

In [21], Brown and Fernandez describe a single security pattern, the Authenticator, that describe a general mechanism for providing identification and authentication to a server from a client. Although authentication is a very important feature of secure systems, the pattern, as described, was limited to distributed object systems. Fernandez and Warrier, in [22], propose a generalization of the latter pattern by offering a single entry point that proxies the authentication interaction with the proper server.

Braga et al. [14] investigate security-related patterns specialized for cryptographic operations. They show how cryptographic transformations over messages could be structured as a composite of instantiations of the cryptographic meta-pattern. This meta-pattern would effectively allow designers to apply cryptography in their applications, permitting to transparently change the underlying cryptographic operations.

The Open Group [11] has possibly introduced the most mature design patterns up-to-date. Their catalog proposes 13 patterns, and these are based on architectural framework standards. The catalog is supplemented by an example of the patterns' use for solving a secure mail problem.

Priebe et al. [56] describe a pattern language for access control that describes how to design role-based access control, discretionary access control and variants of metadata-based access control patterns.

The field has also seen the emergence of what may be a core reference similar to the "Gang of Four" patterns [23] in typical software design. This book, written by Schumacher

et al. [60] updates many of the patterns presented previously, conforming them to the authors' template, and presents other patterns that are mostly at the architectural level.

2.2.2 Evaluation

The design principles and patterns for secure systems are quite sound but are not meant for maintainers, although they could be useful guides for a system redesign. However, the basic principle of the pattern is promising and should be leveraged in order to clearly specify how security hardening is to be done.

Patterns are currently written in a way that requires manual adaptation, and can be written in a way that make their usefulness limited. This seems more like a limitation of the expression format, and not of the concept itself. This conclusion brought us to adopt the idea of security hardening patterns as a way to specify security hardening, while also considering a new form of pattern expression that we describe in Chapter 4.

They also have other shortcomings that we now discuss. In our research published in [40], we examined contributions on security design patterns that were introduced by most of the proposals we discussed. Although not all were described as security design patterns per se, those references all included patterns that could be used in design activities and were thus considered as such. We identified recurring issues in the patterns studied. The patterns were manifesting one or many of the following undesirable characteristics. They were over-specified, under-specified, lacking generality, lacking consensus or misrepresented. We now describe these undesirable characteristics.

Over-Specified Patterns: An over-specified pattern is one for which the specification provides more details and properties than needed. Patterns with variants are considered to be over-specified, as each variant would lead to a different pattern. In all cases, the understandability of the pattern is impacted, as too many details will inevitably create confusion.

Under-specified Patterns: An under-specified pattern is a pattern that is incomplete, or that sees its specification not having enough properties. This will cause understanding or implementation problems, especially if no structure is provided in the specification.

Patterns Lacking Generality: Kienzle et al. [36] classified patterns in four levels of abstraction (1) concept, (2) classes of patterns, (3) patterns and (4) examples. A pattern lacking generality is at a level 4 of abstraction, but it is claiming to be at level 3. This means that it is not general enough to allow it to be used in multiple circumstances with multiple targets [36]. Most of the patterns we studied were at level 4, as they were very practically-oriented, and often proposed with a single context in mind.

Patterns Lacking Consensus: A pattern lacking consensus can be recognized when its solution and/or intent is not uniform in the literature. Patterns are also lacking consensus when named using terminology common in the literature, but for which a different concept is associated. This makes the pattern of little use, as developers and maintainers are likely not to share a common understanding about the system's design.

Misrepresented Patterns: A misrepresented pattern is a pattern that does not offer what it claims to, especially if its naming or intent have little to do with the rest of the pattern.

2.3 Security via Aspect-Oriented Programming

AOP offers a new approach for software development allowing the separation of different concerns into different software units called aspects. Aspects are then combined to software in a manner such that the resulting software is augmented by the behavior specified in the aspects. Some researchers in this emerging field of research have shown that this approach could be used to improve software security in a manner that is more concrete and more systematic than the other proposals we discussed so far. But, before examining the existing research in the field of AOP-based security, let us first survey the field of AOP.

2.3.1 Aspect-Oriented Programming Models

Aspect-Oriented Programming is a family of approaches that allow the integration of different concerns into usable software in a manner that is not possible using the classical object-oriented decomposition. The AOP philosophies are divided between symmetrical and asymmetrical approaches. The symmetrical approaches allow the integration of different models with each other, which consists of the multidimensional separation of concerns and the composition filters. The asymmetrical approaches are adaptive programming and the pointcut-advice models. These offer the enhancement of a model by the aspects. We now briefly see those models.

Multidimensional Separation of Concerns: The multidimensional separation of concerns (also known as hyperspaces [54]) reshapes software in order to facilitate its own evolution, integration and reuse. Hyperspaces support the integration of many models, without the need of a base model as in the asymmetrical approach. Each concern is encapsulated in a

hyperslice and hyperslices can be composed with others in hypermodules.

Composition Filters: Composition filters [6] aim at providing adaptability and reusability via orthogonality. This orthogonality is twofold: Filters are specified with their own interfaces in a language-independent manner and that each filter specification is independent from each other. Filters are able to implement concerns such as inheritance, delegation, synchronization, real-time constraints, and inter-object protocols. This approach defines filters that are applied over method invocations. These can be stacked together, but they are not able to alter the behavior of one another.

Adaptive Programming: Adaptive programming [43,71] is an aspect-oriented approach inspired by the Visitor pattern [23], composed of an adaptive visitor and a traversal strategy. Consequently, the program structure and its behavior are loosely coupled. This approach does not allow the modification of existing program elements, but only allows the augmentation of existing behavior.

Pointcut-Advice Model: The pointcut-advice model for AOP is the most popular in the field. It is so popular, in fact, that it is often a synonym for AOP. Unless specified otherwise, the term AOP in this thesis will refer to this model. This model aims at augmenting a model with multiple aspects. In other terms, it aims at enriching a specific code base.

To develop under this paradigm, one must determine what code needs to be injected into the basic model. Each atomic unit of injection is called an advice. Now that the *what* has been specified, it is necessary to formulate the *where*. This is done by the use of pointcut expressions, its matching criteria restricts the set of a program's join points for which the advice will be injected. The pointcut expressions typically allow one to match

on function calls, function execution, on the control flow ulterior to a given join point, on the membership in a class, etc.

Two sub-problems are present in this approach: Matching and weaving. Matching is the problem of determining if a given pointcut's criteria are are satisfied for a given join point. Certain pointcuts can be determined statically, in the matching process, whereas others can only be determined dynamically, during program execution. Weaving happens upon the determination that an advice's pointcut is matched for a join point and constitutes the injection of the advice's code and possibly of conditional statements for the dynamic matching determination. It is also important to note that some AOP implementations transform aspects into singleton classes, which has an impact on the design of solutions. Leading implementations (often named weavers) of this model are, for Java, AspectJ [35], and for C and C++, AspectC++ [68].

2.3.2 Aspect-Oriented Approaches for Improving Security

AOP appears to be a promising paradigm for software security, since aspects allow us to precisely and selectively define and integrate security objects, methods and events within applications. A few contributions were published on using AOP to inject security code.

Cigital Labs proposed an AOP language called CSAW [63,64], a small aspect-oriented superset of the C programming language. Their work is mostly dedicated to improve the security of C programs. These aspects were divided in both low-level and high-level categories. The low-level aspects target problems such as the exploitation of environmental

variables and format string vulnerabilities. Low-level security aspects also deal with variable verification issues behind buffer overflow attacks, confidentiality issues and communication encryption. Their high-level aspects address the problems of event ordering, signal race condition and type safety.

De Win, in [20], explored the usefulness of two approaches in integrating security aspects within distributed applications. These approaches are interception and Aspect-Oriented Software Development (AOSD). The interception used a coarse-grained mechanism consisting of putting an interceptor at the border of the application, where interactions are checked and approved. This proposition is achieved by changing the software that is responsible of the external communication for the applications. The weaving-based AOSD used AOP to specify the behavior code to be merged in the application and the location where this code should be injected. De Win validated this latter approach by developing some aspects using AspectJ to enforce access control and modularize the addition and access control features of an FTP server.

In [12], Bodkin surveyed the security requirements for enterprise applications and described examples of security crosscutting concerns. This contribution focused on authentication and authorization, while also discussing their related use cases and scenarios. Bodkin explored how such security rules could be implemented using AspectJ, outlining several of the problems and opportunities that arose in applying aspects to secure Java-based web applications. Shlowikowski and Ziekinski discussed in [66] some AspectJ-based security solutions based for J2EE. They explored how the code of security technologies could be injected and weaved in the original application. Those contributions demonstrated the usability of AOP for improving security in large-scale software.

Another contribution in AOP security is the Java Security Aspect Library, in which Huang et al. [32] introduced and implemented a reusable and generic AspectJ library that provides security functions. In order to create a generically usable solution, they used abstract pointcuts, leaving to the programmer the responsibility of defining it in a concrete aspect. Their solution falls short of offering a systematic solution for the improvement of security, as it remains that the programmer must know where the code should be injected.

2.3.3 Security-Related Pointcuts

In the asymmetric AOP approach, the pointcut is a central construct that allows us to determine where advices should be injected. As such, if we need to improve the conditions on which we can inject security code, we need to propose pointcuts that appropriately do so. Many authors have made such contributions, which we will list now.

A dataflow pointcut that is used to identify join points based on the origin of values is defined and formulated by Masuhara and Kawauchi [45] for security purposes. This pointcut permits us to detect if the data sent over the network depends on information from a confidential file.

In [29], Harbulot and Gurd proposed a model of a loop pointcut that shows the need to a loop join point that predicts the occurence of infinite loops, which can be used by attackers to perform denial of service of attacks.

Another approach, that discusses local variables set and get pointcuts, has been proposed by Myers [52]. They would be necessary in order to increase the efficiency of AOP

in security, since they allow the tracking of local variable values inside a method. For instance, confidential data can be protected using such pointcuts by writing advices before and after the use of these variables.

In [13], Bonér discussed a pointcut that is needed to detect the beginning of a synchronized block and add some security code that limits the CPU usage or the number of instructions executed. This usefulness applies also in the security context and can help in preventing many denial of service attacks.

A predicted control flow (`pcflow`) pointcut was introduced by Kiczales in a keynote address [34] without a precise definition. Such pointcut may allow to select points within the control flow of a join point starting from the root of the execution to the parameter join point. The same presentation also introduced an operator allowing to obtain the minimum of two `pcflow` pointcuts, but never clearly defined what this `min` can be or how it can be obtained. These proposals could be used for software security, in the enforcement of policies that prohibit the execution of a given function in the context of the execution of another one.

2.3.4 Evaluation

We evaluated our needs and saw that we need to use a technology that enables radical transformation over the original source code, as security improvements can require changing what is available, augmenting it, or even removing it. Colleagues in the Computer Security Laboratory showed that the pointcut-advice model was the most appropriate for security hardening [2] by analyzing different AOP models as well as their implementations.

The symmetrical approaches were of limited use because of the natural asymmetry inherent to software security enhancements. Furthermore, Hyper/J is limited to method-level granularity. Composition filters are also too limiting for our needs, as security improvements require behavior changes beyond that filters can offer. Adaptive programming was the least useful approach for our needs, since we need more flexibility than adding capabilities to the software. In the case of the pointcut-advice model, we saw that this approach is well-suited for our needs in both its model and in the tools implementing it.

We also see that many features useful for security using AOP have been found missing by authors in the field, resulting in many proposals. Still, the corpus of research in AOP-based security still falls short of an entirely systematic solution usable by non-experts.

2.4 Summary

We saw in this chapter that three major approaches exist for software security: Secure programming practices, secure system design (possibly using patterns) and AOP. We have seen that secure programming practices are too reliant on one's sagacity to be used for systematic security improvements. We also observed that security design patterns tend to be either too abstract or too concrete in order to be really useful for non-experts. In the case of AOP, we saw that this approach offers strong potential for automatically injecting security code, but that it is not yet offering the ability to systematically specify security improvements.

Chapter 3

Security Hardening Practices

The size and complexity (and sometimes lack of documentation) of the code base of a legacy system typically justify a "minimal effort" approach to software maintenance, and results in difficult and expensive changes when serious transformations must be put in place. As such, it is difficult to gather the resources necessary to proactively improve the security level of a legacy application. This is why it is attractive to determine how to systematically perform the desired improvements, so that tools can support (and possibly automate) the process.

A precondition for this systemization is to know what precisely constitutes security hardening and the ways it can be performed. To do so, we chose to examine the literature on secure programming, security engineering and performed a study on the security improvements within the Linux kernel. We focused on C secure programming due to the

language's popularity in production-level software[1] and for the large body of research already performed on this topic. We studied the Linux kernel (versions 2.4 and 2.6) because it is the most popular open source operating system kernel[2].

In Section 3.1, we define security hardening. Then, in Section 3.2, we introduce a high-level approach to security hardening based on an ontology. In Section 3.3, we elaborate the methods of hardening against vulnerabilities related to C programs, structured according to our classification. Afterwards, we study the security hardening practices employed in the Linux kernel in Section 3.4 before summarizing this chapter in Section 3.5.

3.1 Security Hardening Taxonomy

Security hardening at the application level is a relatively unknown term in the current literature and, as such, we defined it in [47]. We also proposed a taxonomy of security hardening methods that refer to the areas to which the solution is applied. We established our taxonomy by studying the solutions of software security problems discussed by the literature. We also investigated the security engineering of applications at different levels, including specification and design issues [9, 11, 30]. From this information on how to correctly build new programs, hardening advice existing in the literature, and our general security background, we were able to draw out a definition and classification of software hardening methodologies.

[1] according to SourceForge.net statistics, 26.22% of open source projects at "Production" and "Mature" levels are written in C [25]

[2] A recent analysis estimates that Linux-based systems represents 12.7% of server revenues [33]. An older study estimates that systems running on the Linux operating system kernel represent 24% of servers and 3% of desktop systems [28]

We define software security hardening as any *process, methodology, product or combination thereof that is used to directly add security functionalities, remove vulnerabilities and/or prevent their exploitation in existing software.* This definition focuses on vulnerability correction, not on their detection. In this context, the following constitutes the detailed classification of security hardening methods:

Code-Level Hardening: Code-Level hardening constitutes *changes in the source code in a way that prevents vulnerabilities without altering the design.* It is essentially the removal of vulnerabilities by retroactively applying proper coding standards that were not enforced originally.

Software Process Hardening: Software Process hardening is the *addition of security features via software build process without changes in the original source code.* It mainly revolves around the use of compiler options, but also includes static library linking with more secure versions of the library.

Design-Level Hardening: Design-Level hardening is the *re-engineering of the application in order to integrate security features that were absent or insufficient.* It refers to changes in the application design and specification. Some security vulnerabilities cannot be resolved by a simple change in the code or by a better environment, but are due to a fundamentally flawed design or specification. Changes in the design are thus necessary to solve the vulnerability or to ensure that a given security policy is enforceable. Moreover, some security features need to be added for new versions of existing products. This category of hardening practices targets more high-level security such as access control, authentication and secure communication. It also includes the use of a more appropriate platform in order to

support the security features in the case of software that exhibits strong dependencies with its platform.

Operating Environment Hardening: Operating Environment hardening consists of *improvements to the security of the execution context that is relied upon by the software.* It impacts the security of the software in a way that is unrelated to the program itself. This addresses the operating system (typically via configuration), the protection of the network layer, the use of high-security dynamically-linked libraries, the configuration of middleware, the use of security-related operating system extensions, the normal system patching, etc.

3.2 Hardening for High-Level Security

This section illustrates our proposition and methods to harden high-level security into applications. At that level, deploying such type of security is mainly categorized as design-level hardening. For this reason, some methods of security design and application re-engineering will be needed to achieve our goal. Our approach for hardening of high-level security uses a security ontology. Ontologies are the specification of a conceptualization of a knowledge domain [59]. A security ontology provides definition of security concepts and relations between them. This is useful due to the existence of different definitions in security literature, and it clarifies the relationships between the elements, improving the understanding of the overall security concept. The security ontology used here is part of a complete one presented in [59]. We chose this ontology because it describes a generic model of security

applied in all the domains. As such, these definitions can be modified to fit in other specific application domains. Figure 3.1 illustrates those concepts and shows the relationships among them.

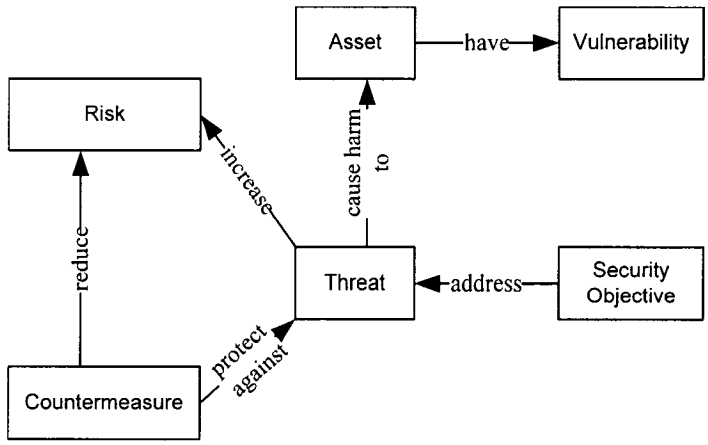


Figure 3.1: Security Ontology Architecture

The following is a brief description of this security ontology's components and their relationships:

Assets: Assets are information or resources that have value to an organization or person. Applications, systems and networks are counted as assets. The weak assets are those that have vulnerabilities.

Vulnerabilities: Vulnerabilities are flaws or weaknesses in an asset. Attackers exploit vulnerabilities to break the security of an asset.

Risks: Risks are determined by taking in consideration the likelihood of a successful attack to an asset and the severity of its impact. Threats increase the risk of a security breach, while countermeasures reduce it.

Security Objectives: Security objectives are an abstract statement of the desired security

level to be achieved.

Threats: Threats are exploitable vulnerabilities. Threats cause harm to assets and increase the risk of security breach.

Countermeasures: Countermeasures are mitigation techniques performed in order to protect an asset against threats and attacks. Countermeasures reduce the risk of security breach.

Expressing the security concept in such an ontological way permits us to better identify and order the steps needed to harden security into applications. This ontology can be converted into a process, where each entity dependent on another becomes an analysis step dependent on the results of the previous related step(s). Although security hardening focuses on finding the best countermeasure to a particular threat, we have no choice to perform some prerequisite tasks in order to achieve a complete hardening process. In this context, we present in the following subsections the steps needed to harden security into applications.

3.2.1 Identifying Threats and Calculating Risks

Identifying threats is an important task in security hardening since we need to determine which threats require mitigation and how to mitigate them, preferably by applying a structured and formal mechanism or process. As such, the following is a brief description of the three main steps needed to identify and evaluate the risk underlying a threat:

Application Decomposition: Application Decomposition is dividing the application into its key components to identify the trust boundaries between them. This decomposition helps

to minimize the number of threats that need mitigation by excluding those that are outside the scope and beyond the control of the application.

Threat Identification: Threat Identification is mainly the categorization of threats with respect to the STRIDE classification [30, 31]: Spoofing identity, tampering with data, repudiation, information disclosure, denial of service and elevation of privilege.

Risk Evaluation: Risk Evaluation consists of the prioritization of threats to be mitigated, based on the likelihood and impact of exploitation.

3.2.2 Countermeasures

Once the previous steps are done and the threat is well identified and categorized, it is possible to determine the appropriate technique(s) to mitigate it. In the literature, it is possible to find a mapping between categories of threats and known counter-measures addressing them. Choosing the best techniques will be mostly based on the state of the art of weaknesses and mitigation methods as well as security patterns. For instance, in [30], the authors provide a list of mitigation techniques for each category of threats of their classification, an excerpt of which we show in 3.1.

Threat Type	**Mitigation Techniques**
Spoofing Identity	Appropriate authentication, protect secret data
Tampering with Data	Appropriate authorization, hashes, message authentication codes, digital signatures
Repudiation	Digital signatures, timestamps, audit trails
Information Disclosure	Authorization, Encryption, protect secrets
Denial of Service	Appropriate authentication and authorization, filtering, throttling, quality of service
Elevation of Privilege	Run with least privilege

Table 3.1: Mapping Between Threats and Mitigation

3.3 Hardening of Low-Level Security

We summarize in this section the major safety vulnerabilities of C programming that we described in Section 2.1 and published in [47]. Table 3.2 shows common solutions to buffer overflows, whereas Tables 3.3 and 3.4 detail solutions for integer and memory management vulnerabilities, respectively.

Hardening Level	**Product/Method**
Code	Bound-checking, memory manipulation functions with length parameter, null-termination, ensuring proper loop bounds, format string specification, input validation
Software Process	Compile with compiler safeguards such as canary words [19]
Design	Input validation, input sanitization
Operating Environment	Disable stack execution, enable stack randomization, use `libsafe` [5]

Table 3.2: Hardening for Buffer Overflows

Hardening Level	**Product/Method**
Code	Use of functions detecting integer overflow/underflow, migration to unsigned integers, ensuring integer data size in assignments/casts
Software Process	Compiler option to convert arithmetic operation to error condition-detecting functions

Table 3.3: Hardening for Integer Vulnerabilities

Hardening Level	**Product/Method**
Code	`NULL` assignment on freeing and initialization, error handling on allocation, pointer initialization, avoid null dereferencing
Software Process	Using aspects to inject error handling and assignments, compiler option to force detection of multiple-free errors
Operating Environment	Use a hardened memory manager (e.g. dmalloc, phkmalloc)

Table 3.4: Hardening for Memory Management Vulnerabilities

3.4 Security Hardening in the Linux Kernel

Some studies on the security of the Linux kernel done in the past incorporated useful statistics on the preponderance of certain types of software vulnerabilities [17, 61, 74]. These studies, however, provided little information about the security improvements that were done as a response to these vulnerabilities. As such, we studied the methods used to improve the security in the Linux operating system kernel. We found and analyzed the patches resolving the vulnerabilities listed in 196 security advisories. Our major contribution is a classification of the security solutions used for the improvement of the Linux kernel, as well as statistics on their relative importance. After detailling our methodology (see 3.4.1), we show our results (see 3.4.2).

3.4.1 Methodology

We chose to use the National Vulnerability Database [53] as our source of vulnerabilities for its official nature and comprehensive contents. Using its online querying system, we were able to extract 196 unique vulnerability advisories (numbered with the CVE/CAN – YYYY – NNNN pattern) for the 2.4 and 2.6 Linux kernel versions.

The second step of our research was to track the precise patch that solved each vulnerability. In some cases, the advisory would include a link to the revision control systems used by the Linux project (GIT and BitKeeper [10]), mailing list archives containing the patch or a vendor's advisory or bug-tracking entry that contained the patch. When such information was unavailable, we resorted to examine unaffected version's changelogs, vendors' bug tracking systems, maintainer mailing lists, maintainers' management systems,

enthusiasts' websites and newsgroups. This information often lead us, directly or indirectly, to the patch for the given vulnerability. For instance, a changelog file would include the CVE/CAN number, or have a comment that allowed us to conclude that a given change was related to the vulnerability investigated. The commit comment or GIT commit number was then queried for in the BitKeeper archives, enabling us to retrieve the patch easily.

As we were finding the vulnerability solutions, we also examined them and noted a summary of the code changes we observed. Broad categories quickly emerged as we were doing this activity, and we started allocating the solutions to their matching category.

3.4.2 Results

While gathering and observing our data, we grouped it in 13 categories of solutions that were used by Linux kernel maintainers: Change of data types, precondition validation, ensuring atomicity, error handling, zeroing memory, freeing resources, input validation, capability validation, fail-safe default initialization, protection domain enforcement, redesign, other, and unresolved. In our analysis, we observed that error handling, redesign, and precondition validation were dominating the solutions (please consult Table 3.5). Please note that the numbers purposefully do not match the number of advisories evaluated (196 CVEs), as many methods can be used to solve the vulnerabilities in one advisory.

In order to understand these categories, we now define and illustrate each remedial method via a patch. The results are taken directly from the patches associated to different CVEs, expressed in the patch file format [44].

Remedial Types	Count	%
Error Handling	56	23.24
Redesign	44	18.26
Precondition Validation	37	15.35
Fail-Safe Default Initialization	17	7.05
Other	16	6.64
Change of Data Types	14	5.81
Ensuring Atomicity	12	4.98
Zeroing Memory	11	4.56
Input Validation	10	4.15
Protection Domain Enforcement	8	3.32
Capability Validation	7	2.90
Freeing Resources	5	2.07
Unresolved	4	1.66
Total	241	100.00

Table 3.5: Vulnerability Solution Distribution

Change of Data Types: A change of data types is *the use of more appropriate data types when necessary.* In many cases, the data type used was not appropriate for the data representation needed or the concurrency scenario. Typically, the change is from signed to unsigned types, or from static to non-static. The Listing 3.1 shows how a vulnerability was remedied by making a variable non-static, which insures that it cannot be shared between threads.

```
  enum ip_nat_manip_type maniptype,
  const struct ip_conntrack *conntrack)
  {
-   static u_int16_t port, *portptr;
+   static u_int16_t port;
+   u_int16_t *portptr;
    unsigned int range_size, min, i;
```

Listing 3.1: Example of Change of Data Type by Removing Static Assignment for CVE – 2005 – 3275

Precondition Validation: Precondition validation is *the ensuring that an operation's set of preconditions are met before proceeding with its execution.* This is typically done by checking for valid variable values and a valid range, then returning an error code if the

conditions are not met. Additionally, this may include the improvement of an existing precondition check. Listing 3.2 shows how a vulnerability was remedied by ensuring the validity of the `len` parameter.

```
  count = depth->n;
+ if( count > 4096 || count <= 0 || count * sizeof(*x) <= 0 ||
+    count * sizeof(*y) <= 0)
+    return -EMSGSIZE;

  x = kmalloc( count * sizeof(*x), 0 );
```

Listing 3.2: Example of Improved Precondition Validation for CVE – 2004 – 0003

Ensuring Atomicity: Ensuring atomicity is a *code modification that guarantees non-concurrent execution of critical sections.* As such, a maintainer must ensure proper locking of critical sections before executing them (or merely obtaining a reference to sensitive data structures), as well as ensuring the proper management of timers, when applicable. Listing 3.3 shows how a vulnerability can be remedied by ensuring the locking of a data structure before manipulating it through a pointer.

```
  struct ebt_chainstack *cs;
  struct ebt_entries *chaininfo;
  char *base;
- struct ebt_table_info *private = table->private;
+ struct ebt_table_info *private;

  read_lock_bh(&table->lock);
+ private = table->private;
  cb_base = COUNTER_BASE(private->counters, private->nentries,
     cpu_number_map(smp_processor_id()));
  if (private->chainstack)
```

Listing 3.3: Example of Atomicity Guarantee for CVE – 2005 – 3110

Error Handling: Error handling is *the addition or improvement of the verification and handling of error conditions.* In many cases, the error status of some functions is not checked by the calling, or the handling of the error is not appropriate. Error handling is

most easily distinguished from precondition validation by the fact that it typically occurs within the body of the function, whereas precondition validation generally occurs at the beginning of a function. Listing 3.4 shows how an error condition requires memory freeing and the communication of an error status to the caller.

```
@@ -57,6 +57,7 @@ int syscall32_setup_pages(struct linux_binprm *bprm,
    int exstack)
         int npages = (VSYSCALL32_END - VSYSCALL32_BASE) >> PAGE_SHIFT;
         struct vm_area_struct *vma;
         struct mm_struct *mm = current->mm;
+        int ret;

         vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
         if (!vma)
@@ -78,7 +79,11 @@ int syscall32_setup_pages(struct linux_binprm *bprm,
    int exstack)
         vma->vm_mm = mm;

         down_write(&mm->mmap_sem);
-        insert_vm_struct(mm, vma);
+        if ((ret = insert_vm_struct(mm, vma))) {
+                up_write(&mm->mmap_sem);
+                kmem_cache_free(vm_area_cachep, vma);
+                return ret;
+        }
         mm->total_vm += npages;
         up_write(&mm->mmap_sem);
         return 0;
```

Listing 3.4: Example of Improved Error Handling for CVE – 2005 – 2617

Zeroing Memory: Some sensitive information can be left in memory from previous kernel operations, allowing the possibility of private data leaks. Zeroing Memory, defined as *the overwriting of selected memory contents by it with filling zero values*, is the preferred solution to this problem. Listing 3.5 shows how a disclosure vulnerability was solved by using `memset`.

Freeing Resources: Some security vulnerabilities can occur due to memory leaks or unreleased resources such as timers and dynamic memory segments. Freeing resources is

```
+    memset(&info, 0, sizeof(info));
     strncpy(info.id, "USB_AUDIO", sizeof(info.id));
     strncpy(info.name, "USB Audio Class Driver", sizeof(info.name));
     if (copy_to_user((void *)arg, &info, sizeof(info)))
```

Listing 3.5: Example of Zeroing Memory to Solve Data Leaks in CVE – 2004 – 0685

ensuring that all resources are freed after usage. Listing 3.6 shows how a denial of service condition was remedied by removing memory leaks.

```
      kenter("{%d}", key->serial);

      key_put(rka->target_key);
+     kfree(rka);

 } /* end request_key_auth_destroy() */
```

Listing 3.6: Example of Remediation of Memory Leaks for CVE – 2005 – 3119

Input Validation: Input validation is *the practice of validating or modifying input data in order to ensure it complies with a safety policy.* This is done normally by truncation or filtering and is highly important when user-supplied data traverses a security boundary. Listing 3.7 shows how input validation capabilities were not fully used for bridge forwarding functions, which created a vulnerability, and how said vulnerability was eliminated by ensuring that a specific subset of traffic was filtered.

```
  struct net_bridge *br = p->br;
  unsigned char *buf;

+ /*insert into forwarding database after filtering to avoid spoofing*/
+ br_fdb_update(p->br, p, eth_hdr(skb)->h_source);
+
  /* need at least the 802 and STP headers */
  if (!pskb_may_pull(skb, sizeof(header)+1) ||
      memcmp(skb->data, header, sizeof(header)))
```

Listing 3.7: Example of Improved Input Validation Remediating CVE – 2005 – 3272

Capability Validation: Capability validation is *the verification that execution rights are granted before performing an operation.* This is a special case of precondition validation, which we considered worthy to mention separately. Listing 3.8 shows how a simple check remedies a security vulnerability.

```
do_kdgkb_ioctl(int cmd, struct kbsentry
  int i, j, k;
  int ret;

+ if (!capable(CAP_SYS_TTY_CONFIG))
+   return -EPERM;
+
  kbs = kmalloc(sizeof(*kbs), GFP_KERNEL);
  if (!kbs) {
    ret = -ENOMEM;
```

Listing 3.8: Example of Capability Validation Solving CVE – 2005 – 3257

Fail-Safe Default Initialization: Fail-safe default initialization is *the setting of a guaranteed safe value for variables before an operation requiring its use is performed.* This includes the assignment of a default value at declaration time and ensuring null-termination of strings. Listing 3.9 shows how setting a pointer to NULL remedied a denial of service vulnerability.

```
  size_t array_size;

  /* set an arbitrary limit to prevent arithmetic overflow */
- if (size > MAX_DIRECTIO_SIZE)
+ if (size > MAX_DIRECTIO_SIZE) {
+   *pages = NULL;
    return -EFBIG;
+ }

  page_count = (user_addr + size + PAGE_SIZE - 1) >> PAGE_SHIFT;
  page_count -= user_addr >> PAGE_SHIFT;
```

Listing 3.9: Example of Fail-Safe Default Initialization Remediating CVE – 2005 – 0207

Protection Domain Enforcement: Protection Domain Enforcement is *guaranteeing that the data operated on, or the execution flow, is in the appropriate security domain.* In the case of the Linux kernel, the data can be in either user space or in kernel space. The kernel implements this differentiation by copying data between kernel and user space as needed, hereby ensuring that the information is not altered during an operation. Listing 3.10 shows an example of this, where the call `copy_to_user` was added with error handling in order to enforce the protection domain.

```
case VIDIOCSWIN:
{
-   struct video_window *vw = (struct video_window *) arg;
-   DBG("VIDIOCSWIN %d x %d\n", vw->width, vw->height);
+   struct video_window vw;

-   if ( vw->width != 320 || vw->height != 240 )
+   if (copy_from_user(&vw, arg, sizeof(vw)))
+     {
        retval = -EFAULT;
+       break;
+     }
+
+   DBG("VIDIOCSWIN %d x %d\n", vw->width, vw->height);

+   if ( vw.width != 320 || vw.height != 240 )
+     retval = -EFAULT;
      break;
    }
```

Listing 3.10: Example of Protection Domain Enforcement Remediating CVE – 2004 – 0075

Redesign: We define redesign in terms of significant code changes, from refactoring to the change of a provider package, including the introduction of new or modification of the existing APIs. It is a family of major changes more than a specific solution type. Listing 3.11 shows how a dangerous option needed to be removed from the code.

Other: Other methods were not encountered in a frequency allowing us to classify them

```
- sctp_lock_sock(sk);
-
- switch (optname) {
- case SCTP_SOCKOPT_DEBUG_NAME:
- /* BUG! we don't ever seem to free this memory. --jgrimm */
- if (NULL == (tmp = kmalloc(optlen + 1, GFP_KERNEL))) {
-   retval = -ENOMEM;
-   goto out_unlock;
- }
-
- if (copy_from_user(tmp, optval, optlen)){
-   retval = -EFAULT;
-   goto out_unlock;
- }
- tmp[optlen] = '\000';
- sctp_sk(sk)->ep->debug_name = tmp;
- break;
+ sctp_lock_sock(sk);

+ switch (optname) {
  case SCTP_SOCKOPT_BINDX_ADD:
```

Listing 3.11: Example of Redesign by Removal of Option for CVE – 2004 – 2013

into category for themselves. We chose a threshold at three instances, since we considered that solutions encountered less than 1% of the time are unlikely to be common practices. As such, we chose to group these cases in this category. Listing 3.12 shows a vulnerability solved by changing default file permissions.

```
 MODULE_PARM_DESC(debug, "Enable debug output");

 module_param_named(cards_limit, drm_cards_limit, int, 0444);
-module_param_named(debug, drm_debug, int, 0666);
+module_param_named(debug, drm_debug, int, 0600);

 drm_head_t **drm_heads;
 struct drm_sysfs_class *drm_class;
```

Listing 3.12: Example of Other Solution: Changing Default File Permissions for CVE – 2005 – 3179

Unresolved: Unresolved flaws are flaws for which a solution was not available when the research is performed.

3.5 Summary

In this chapter, we defined security hardening, and showed how it was practically performed. We also saw security hardening classification that emerged from the state of the art in the field. At a higher level, we saw how an ontology of vulnerabilities could also serve as process in order to identify the countermeasures to implement. At a lower level, we saw different hardening approaches for solving buffer overflows, integer and memory management vulnerabilities. Then, we showed the results of our study on the security improvements in the open source Linux kernel, which allowed us to establish a classification of these practices. Overall, we were able to increase our understanding of the methods used in the industry in order to increase the security level.

Chapter 4

An Aspect-Oriented Approach to Security Hardening

Now that we have studied the nature of security hardening and the practices employed to perform it, we now inquire into how security hardening, at the code and design levels, can be performed in a systematic manner.

This chapter presents our proposed approach for systematic security hardening, which we introduced in [50, 51]. Each component participates by playing a role and/or providing functionalities in order to have a complete security hardening process. The primary objective of this approach is to allow the developers to perform security hardening of FOSS by applying proven solutions without the need to have expertise in the security solution domain. This is done by providing an abstraction over the actions required to improve the security of the program and adopting AOP to build and develop our solutions. The developers are able to specify the hardening plans that use and instantiate the security hardening patterns using *SHL*.

Our approach (illustrated in Figure 4.1, with the parts grayed being completed in the scope of this thesis) is structured around security hardening plans and security hardening patterns, which can be combined and refined into aspects used to harden a program. The combination of hardening plans and patterns constitutes the concrete security hardening solutions. It constitutes a bridge that allows security experts to provide the best solutions to particular security problems with all the details on how and where to apply them, and allows the software engineers to use these solutions to harden FOSS by simply specifying and developing high-level security hardening plans that are abstract enough to alleviate their need to master several security technologies.

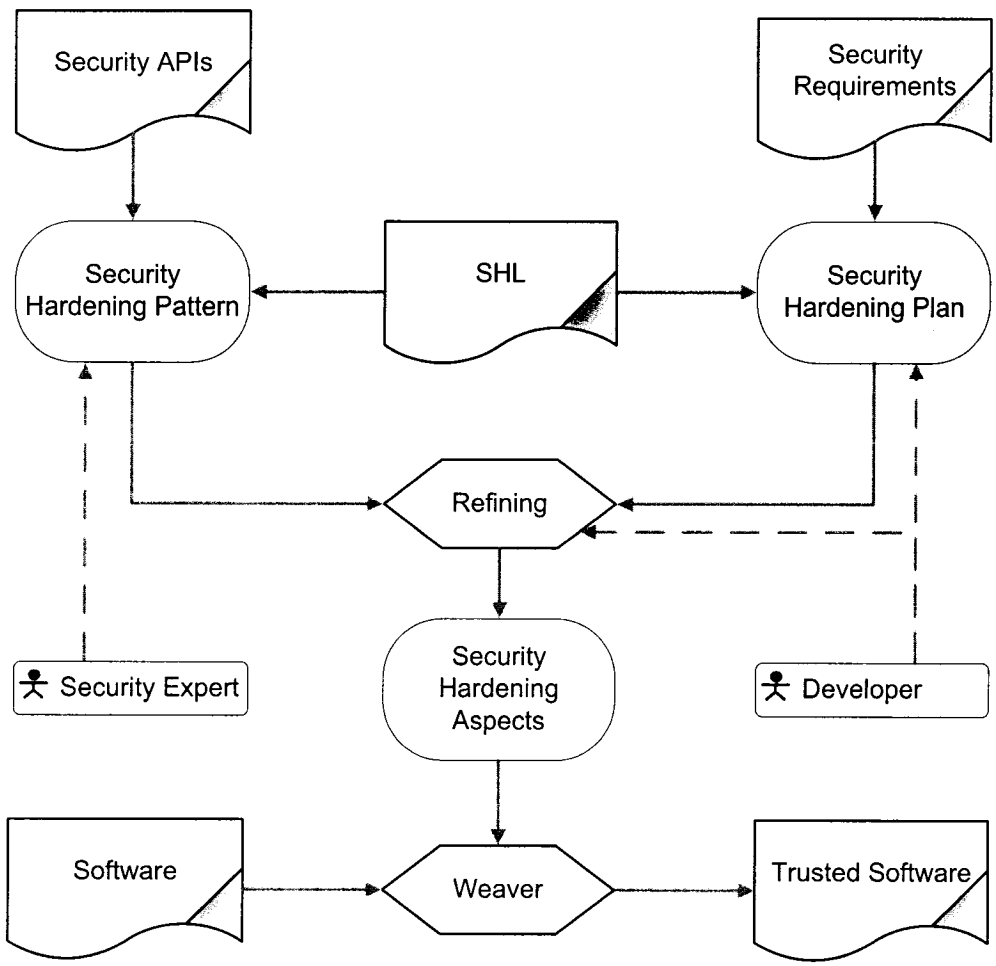


Figure 4.1: Schema of Our Approach

The approach requires that security experts write a variety of security hardening patterns. Those patterns are built upon security APIs and are described in more detail in Section 4.1. The developer only needs to specify hardening plans in accordance with the security requirements. These hardening plans are parametrized patterns specifying the point of application of the pattern and are described in Section 4.2. We then describe the *SHL* language in Section 4.3. We describe quickly the refining step in Section 4.4, which constitutes the translation from *SHL* of a pattern instance into the target language. We experimentally validated our approach early on, and we show our results in Section 4.5. These experiments consist of adding secure connections, authorization and memory encryption to demonstration programs. We finally summarize our findings in Section 4.6.

4.1 Security Hardening Plans

A security assessment brings any decision-maker to perform a risk analysis, which will finally determine the security requirements. A given set of security requirements may be implemented through different combinations of mechanisms and software improvements. As such, a software developer must select the combination of solutions deemed optimal for the specific program to harden (c.f. Section 2.2). This decision is written in a security hardening plan, effectively translating such requirements into a specification of software modifications.

The plans are written as a list of parametrized patterns with a `Where` clause, which indicates where in the software the pattern is to be applied. Each pattern is responsible to document the parameters it requires and supports.

The developer is able to write the different hardening plans that are required for each part of the software (e.g. a server program written in C for the UNIX environment vs. a Java client) and submit them to be refined.

4.2 Security Hardening Patterns

We define security hardening patterns as proven solutions to known security problems, together with detailed information on how and where to inject each component of the solution into the application. Security hardening patterns specify the steps and actions needed to harden systematically security into the code. In this context, security hardening patterns are defined as proven solutions to known security problems, together with detailed information on how and where to inject each component of the solution into the application.

In order to deal with the abstraction level problem that we saw in Chapter 2, we define the patterns in a twofold manner. First, we document the pattern in an abstract manner similar to the current approach to pattern documentation called *abstract pattern specification*. This allows the non-experts to understand the solution provided by the pattern and its applicability. Then, we also define the pattern in a language, platform and technology-dependent manner using *SHL*, named the *practical pattern specification*. We can offer multiple such definitions under the same abstract pattern specification, allowing developers to choose which option better fits their needs. The patterns, as well as the hardened software resulting from our approach, can be verified via static and dynamic methods against security vulnerabilities.

4.3 The *SHL* Language

The main components of our approach are the security hardening plans and patterns that provide an abstraction over the actions required to improve the security of a program. They should be specified and developed using an abstract, programming language-independent and aspect-oriented language. The current aspect-oriented languages, however, are typically programming language-dependent, from which the need to elaborate a language providing the missing features.

Our proposed language, *SHL* [48], allows for the description and specification of security hardening patterns and plans. It is a compact language built on top of the current AOP technologies that are based on the pointcut-advice model. We chose to use this approach in order to facilitate the refining of the patterns specified in *SHL* into aspects of target AOP language. We further developed part of *SHL* with notations and expressions close to those of the current AOP languages, but with the addition of the features we developed in the scope of this research. We implemented this language specification using the ANTLRWorks 1.0b11 development environment [55] and were also able to validate the syntax of different plan and pattern examples within this tool.

4.3.1 Grammar and Structure

In this section, we present the syntactic constructs and their semantics in *SHL*. Figure 4.2 shows the BNF grammar of *SHL*. It can be used for both plans and patterns specification, with a specific template structure for each of them.

```
Start                       ::=  SH_Plan
                            |    SH_Pattern

SH_Plan                     ::=  Plan Plan_Name
                                 SH_Plan_Code
Plan_Name                   ::=  Identifier
SH_Plan_Code                ::=  BeginPlan
                                 Pattern_Instantiation*
                                 EndPlan
Pattern_Instantiation       ::=  PatternName Pattern_Name
                                 (Parameters Pattern_Parameter*)?
                                 Where Module_Identification+
Pattern_Name                ::=  Identifier
Pattern_Parameter           ::=  Parameter_Name = Parameter_Value
Parameter_Name              ::=  Identifier
Parameter_Value             ::=  Identifier | Integer | Collection
Collection                  ::=  { (Identifier | Integer) (, Identifier | Integer)* }
Module_Identification       ::=  Identifier | File
File                        ::=  Identifier . Identifier

SH_Pattern                  ::=  Pattern Pattern_Name
                                 Matching_Criteria?
                                 SH_Pattern_Code
Matching_Criteria           ::=  Parameters Pattern_Parameter+
SH_Pattern_Code             ::=  BeginPattern
                                 Location_Behavior*
                                 EndPattern
Location_Behavior           ::=  Behavior_Insertion_Point+ Location_Identifier+
                                 Primitive*?
                                 Behavior_Code
Behavior_Insertion_Point    ::=  Before
                            |    After
                            |    Replace
Location_Identifier         ::=  FunctionCall <Signature> (Arguments)?
                            |    FunctionExecution <Signature> (Arguments)?
                            |    WithinFunction <Signature> (Arguments)?
                            |    CFlow <Location_Identifier>
                            |    GAFlow <Location_Identifier>
                            |    GDFlow <Location_Identifier>
                            |    ...
Signature                   ::=  Identifier
Primitive                   ::=  ExportParameter <Identifier>
                            |    ImportParameter <Identifier>
Arguments                   ::=  ( Star_Or_Identifier (, Star_Or_Identifier)* )
Star_Or_Identifier          ::=  * | Identifier
Behavior_Code               ::=  BeginBehavior
                                 Code_Statement
                                 EndBehavior
```

Figure 4.2: *SHL* Grammar

Hardening Plan Structure A hardening plan starts always with the keyword `Plan`, followed by its name. The plan is itself contained between a pair of `BeginPlan` and `EndPlan` keywords. Between those, there are one or many pattern instantiations that allow the specification of the name of the pattern and its parameters, in addition to the location where it should be applied. Each pattern instantiation starts with the keyword `PatternName` followed by the name of the pattern, then the keyword `Parameters` followed by a list of parameters and finally by the keyword `Where` followed by the module name where the pattern should be applied (e.g. file name). An example is shown in Listing 6.2.

Hardening Pattern Structure A hardening pattern starts with the keyword `Pattern`, followed by the pattern's name, then the keyword `Parameters` followed by the matching criteria and finally the pattern's code that starts and ends respectively by the keywords `BeginPattern` and `EndPattern`. The matching criteria are composed of one or many parameters that could help in distinguishing between practical pattern specifications with the same name and allow the pattern instantiation. The pattern code is based on AOP and composed of one or many `Location_Behavior` constructs. Each one of them constitutes the location identifier and the insertion point where the behavior code should be injected, the optional primitives that may be needed in applying the solution and the behavior code itself. A detailed explanation of the components of the pattern's code will be illustrated in Section 4.3.2.

4.3.2 Informal Semantics

In this Section, we present the informal semantics of the important syntactic constructs in the *SHL* language.

Pattern_Instantiation Specifies the name of the pattern that should be used in the plan and all the parameters needed for the pattern. The name and parameters are used as matching criteria to identify the selected pattern. The module where the pattern should be applied is also specified in the `Pattern_Instantiation`. This module can be the whole application, file name, function name, etc.

Matching_Criteria This section is used for patterns only, and will define the parameters for which this practical pattern specification is applicable. During the refinement stage, the proper practical pattern specification will be chosen based on these matching criteria.

Location_Behavior Is based on the advice-pointcut model of AOP. It is the abstract representation of an aspect in the solution part of a pattern. A pattern may include one or many `Location_Behavior`. Each `Location_Behavior` is composed of the `Behavior_Insertion_Point`, `Location_Identifier`, one or many `Primitive` and `Behavior_Code`.

Behavior_Insertion_Point Specifies where the code injection should take place. The `Behavior_Insertion_Point` can have the following three values: `Before`, `After` or `Replace`. The use of `Before` or `After` keeps the old code at the identified location and inserts the new code before or after it, respectively. The `Replace` construct

is similar to the `around` found in many AOP languages. It replaces the code identified in the pointcut, unless the advice includes `proceed`, in which case it acts as a wrapper. We do not specify a `proceed` primitive, however, because it would be included in the `Behavior_Code` segment in the language of the underlying AOP technology.

Location_Identifier Identifies the join point or series of join points in the program where the changes specified in the `Behavior_Code` should be applied. It is equivalent to a pointcut in the popular AOP languages. They can be logically combined using boolean connectives as in the existing languages. These constructs should have their equivalent in the current AOP technologies or should be implemented into the weaver used, or the refining process should adequately map them into semantically equivalent constructs in the target AOP language. We now describe the semantics of important labels that are used for identifying locations:

- `FunctionCall <Signature>(Arguments)` Provides all the join points where a function matching the signature specified is called. The optional parameter section specifies parameters matched (either ignored with a $\star$ or assigned a name) that can be used within the `Behavior_Code`.

- `FunctionExecution <Signature>(Arguments)` Provides all the join points referring to the implementation of a function matching the signature specified. The optional parameter section specifies parameters matched (either ignored with a $\star$ or assigned a name) that can be used within the `Behavior_Code`.

- `WithinFunction <Signature>(Arguments)` Filters all the join points that are within the functions matching the signature specified. The optional parameter section specifies parameters matched (either ignored with a * or assigned a name) that can be used within the `Behavior_Code`.

- `CFlow <Location_Identifier>` Captures the join points occurring in the dynamic execution context of the join points specified in the input `Location_Identifier`. As in the AspectJ and AspectC++ `cflow`, this pointcut refers to the join point that are in the execution path after `cflow`'s input join points.

- `GAFlow <Location_Identifier>` Returns the closest guaranteed ancestor to the join points, as specified in Section 5.3.

- `GDFlow <Location_Identifier>` Returns the closest guaranteed descendant to the join points, as specified in Section 5.3 5.

The `Location_Identifier` construct can be composed with boolean operators as follows:

- `Location_Identifier && Location_Identifier` Returns the intersection of the join points specified in the two constructs.

- `Location_Identifier || Location_Identifier` Returns the union of the join points specified in the two constructs.

- `! Location_Identifier` Is a negation. It excludes the join points specified in the construct.

Primitive Is an optional functionality that allows us to specify the variables that should be passed between two `Location_Identifier` constructs. The following are the constructs responsible of passing the parameters:

- `ExportParameter <Identifier>` Defined at the origin `Location_Identifier`. It allows us to specify a set of variables defined at the origin `Location_Identifier` and make them available to be exported. It is defined in Section 5.4.

- `Importparameter <Identifier>` It allows us to specify a set of variables and import them from the origin `Location_Identifier` where the `ExportParameter` has been defined. It is defined in Section 5.4.

Behavior_Code May contain code written in any programming language, or even written in English as instructions to follow, depending on the abstraction level of the pattern. The choice of the language and syntax is left to the security hardening pattern developer. However, the code provided should be abstract and at the same time clear enough to allow a developer or tool to refine it into low level code without the need of high security expertise.

4.4 Pattern and Plan Refinement

The refiner is responsible for using the information in the security hardening plan in order to correctly choose the desired practical pattern specification. Once this is done, the specification is instantiated using the plans' parameters. The first step is to insert the parameters' values where specified in the pattern, similar to macros in C programming (a

feature not available in the current version of *SHL*). Afterwards, the refiner must translate the pattern from its initial technology-independent form into an aspect written in a technology-dependant aspect-oriented language.

It is possible to have maintainers who, without having any security expertise, directly use our plans and patterns in order to manually harden source code. However, this approach would not allow any form of automation that the full realization of our approach promises. The refinement of the solutions into aspects can be performed using a dedicated tool or by programmers. Then, an existing AOP weaver (e.g. AspectJ, AspectC++) can be executed to harden the aspects into the original source code, which can now be inspected for correctness.

4.5 Early Experimental Validation of the Approach

We demonstrated the feasibility of our approach by developing examples that deal with security requirements such as securing a connection, authorization and encrypting information in the memory. This phase was crucial to determine the viability of the "Refining" and subsequent parts of our approach. We performed the following steps:

1. We implemented applications that need to be hardened. These applications are briefly described in sections 4.5.1, 4.5.2 and 4.5.3.

2. We wrote hardening plans for the desired security requirements.

3. We added code that enforce the plans with the least changes in the existing code.

4. We extracted this code into security APIs.

5. We developed patterns describing those source code changes.

6. We wrote aspects implementing those patterns.

7. We applied the aspects on the original programs by using available weavers (AspectJ 1.5.2 and AspectC++ 1.0pre3).

8. We tested the resulting programs for functional and security correctness by comparing them to the manually hardened ones.

9. We measured the execution time of programs hardened by both approaches.

We now examine our results from securing a connection, adding access control and encrypting memory locations.

4.5.1 Secure Connection

A first issue is the securing of channels between two communicating parties to avoid eavesdropping, tampering with the transmission or session hijacking. The Transport Layer Security (TLS) protocol is widely used for this task. We thus decided to implement the "Secure Connection" security hardening on a minimalistic HTTP client application, programmed in `C`, using the GnuTLS TLS library [46] and AspectC++.

Our solution took into consideration programming practices found in the industry. The first consideration is that many programs use sockets for both local and remote communication. The second consideration is that different components (i.e. different classes, modules, functions, etc.) may host the connection management (i.e. `connect/close`) and transmission (i.e. `send/receive`) functions. We thus decided to store a data structure in a

hash table when a call to `connect` was meant to be secured. Then, we wrapped around all the other socket-related functions and used a runtime check to determine whether or not the call needed to be replaced or complemented by the ones provided by the SSL/TLS library. If not, the function calls are executed as normal. This required additional effort to develop additional components that distinguish between the functions that operate on secure and insecure channels and export parameters between different places in the application.

The passing of parameters between the different functions needing them was a difficulty, since the GnuTLS data structures are not type compatible with the handle to the socket (an integer). In order to avoid using shared memory directly, we opted for a hash table that uses the socket number as a key to store and retrieve all the needed information. As such, on `connect`, the GnuTLS data structures were saved in the hash table, making the socket marked as being secured. All calls to `send()`, `recv()` and `close()` are modified for a runtime check so that the GnuTLS functions required to perform the transmission and sesssion termination were used only if the socket was secured.

In order to test the performance impact of the hardening, we iterated over many connections where a connection to a remote web server is established, and a few index page's bytes are retrieved. The validity of the hardening was confirmed using the Wireshark packet capture package [39]. In Table 4.1, the 'Unsecured' column shows the execution time for unsecured sockets using the standard socket API. The 'GnuTLS' columns show three approaches: One passing the GnuTLS variables by a hash table ('Hash Table') , another via the procedure's parameters ('Direct') and the last one shows the performance to using AspectC++ to inject code that is functionally equivalent to 'Hash Table' ('via AOP').

Connections	Unsecured	GnuTLS		
		Hash Table	Direct	via AOP
100	0.234 s	15.937 s	15.89 s	15.953 s
500	1.406 s	79.39 s	80.484 s	79.828 s
1000	3.062 s	159.984 s	157.796 s	161.25 s

Table 4.1: Execution Time for Different Hardening Approaches Securing a Connection

We see from the results that there is a significant cost to securing the channel, but that the different approaches have similar overhead. In the context of our experiment, we did not measure the network and server load fluctuations that could have impacted these measurements, since we presumed these to be too minimal to warrant such measurement. In the case of a thousand iterations, we see that the approach using AOP (and the hash table) has an overhead of about 2.2%, compared to 'direct'. This impact on performance is likely due to the overhead of using the hash table and of the multiple function calls added by AspectC++. Although it is likely that improvements in AspectC++ would narrow this gap, we consider those results as supporting our proposition that AOP is an acceptable mechanism for the injection of security hardening code.

4.5.2 Authorization

Access control is a problem of authorizing or denying access to a resource or operation. It requires the knowledge of which principal is interacting with the application, and what are its associated rights.

We have implemented an example of access control by injecting a capability check relying on the Java Authorization and Authentication Service. The rights are specified in a separate policy file. We assume a local login, in this case, and we obtain the user name from the virtual machine. The permissions are specified in the `package.class.function`

format. The injection of the same code was done manually and with an AspectJ aspect. We also verified the functional and security correctness of the hardened application by removing the right in the policy file and observing an exception thrown instead of the execution of the method. Our runtime experiments show the difference in execution time for multiple runs of a security check between a case hardened manually and one hardened using AspectJ, presented in Table 4.2.

Calls	Unsecured	Secured Manually	Secured with AspectJ
1	<1 ms	48.93 ms	97.8 ms
100	<1 ms	357.77 ms	497.3 ms
500	<1 ms	1019.73 ms	1480.77 ms

Table 4.2: Average Execution Time for Different Approaches to Add Authorization

Similarly to our previous results, we see a significant overhead when the security hardening is applied, no matter which method is used to do so. Contrary to our previous observations, however, we see that the use of AspectJ to perform the hardening causes an overhead of about 45% compared to the manual hardening, when we observe the results over 500 iterations. The operation execution time passed from 2.04 ms per iteration to 2.94 ms per iteration, which is too short for the user to notice. We see that improvements to the technology used (whether it is in the AspectJ compiler itself, or in the AspectJ framework used at runtime) are desirable in order to close the gap with the manual hardening approach. Nevertheless, the impact of using AspectJ remains small enough to consider it as an acceptable tool for security hardening.

4.5.3 Encryption of Memory

Although processes should encapsulate all of their internal information in a manner that is opaque to other processes, modern operating systems allow the reading of the entire memory space, making the information stored in memory vulnerable to local memory reading attacks. Furthermore, the UNIX core dump of a crashed process may contain sensitive information. Cryptography offers tools that are useful to protect confidentiality and to detect integrity violations of data.

We faced the problem of communicating the cryptographic information needed between the program locations where encryption and the decryption was performed. Since we had a similar issue for the experiment in Subsection 4.5.1, we also used a hash table to pass the information from one part of memory to another. In order to keep type compatibility, we generate a counter that is different for `short`, `char` and `int` (normally used to represent pointers as well). The encrypted value as well as cryptographic information is stored in a hash table that uses the aforementioned counter as a key. One limitation of this strategy is the limited range of values that can be kept this way, especially for `char` and `short` variables.

We were able to manually harden the application but, contrary to other examples, we were not able to implement an aspect in order to encrypt only one of the buffers in our reference program due a limitation in the current AOP languages for `C`. This limitation is that it is not currently possible to specify a join point over a variable name, as illustrated in Listing 4.1.

```
/*get username*/
fgets(username, FIELD_LEN, stdin);


/*get password*/
fgets(password, FIELD_LEN, stdin);
```

Listing 4.1: Example Code that cannot be Hardened Using Aspects

4.5.4 Discussion

The latter measurements between the performance cost of hardening manually or using aspect-oriented technologies show little overhead in the case of the AOP-generated code. This information, combined with the identical security correctness as for manually hardened program, demonstrates that the refinements of patterns and plans to AOP is a viable method for hardening applications. However, we also found technological limitations that forced us to resort to complicated tricks in order to obtain our functional objective, if at all possible. We noticed that improvements to AspectC++ and AspectJ would have facilitated this task and kept the aspects much lighter and concise.

4.6 Summary

In this chapter, we presented an overview of the aspect-oriented security hardening approach central to this thesis. This approach is built upon security APIs and security hardening patterns supplied by security experts, and user-generated security hardening plans using the patterns. Our approach allows for these plans and patterns to be refined into aspects, which can then be used on the software to harden. This approach allows for the systematic hardening of software, and the development of an automated refiner that could allow our

approach to be automatic.

Our experiments showed that the approach was feasible, but they also showed that existing AOP technologies are lacking features in order to fully perform security hardening.

Chapter 5

Shortcomings of Current AOP Approaches and New Primitives

As we have seen in the previous chapter, the currently available AOP technologies provide a good foundation for our security hardening approach, but are lacking some features. This conclusion was reached by other researchers in the AOP field [13,29,34,45], who proposed complementary constructs to enrich AOP languages for security hardening.

Those lacking features caused different problems. In a given case, we were not able to perform some security hardening activities. Such limitations are forcing us, when applying security hardening practices, to perform various programming gymnastics, resulting in additional modules that must be integrated within the application, at an elevated runtime, memory and development cost. Moreover, the result of applying this coding strategy is of a higher level of complexity from the standpoint of auditing and evaluation.

In the scope of this chapter, we propose improvements to current AOP technologies, by elaborating new pointcuts and primitives needed for security hardening concerns. In this

context, we propose two new control flow based pointcuts to AOP languages that allow the identification of particular join points in a program's control flow graph (CFG). The first proposed pointcut is the *GAFlow*, the closest guaranteed ancestor, which returns the closest ancestor join point to the pointcuts of interest that is on all their runtime paths. The second one is the *GDFlow*, the closest guaranteed descendant, which returns the closest child join point that can be reached by all paths starting from the pointcuts of interest.

Moreover, we propose two new primitives to AOP languages that are also needed for security hardening concerns. These primitives, called *ExportParameter* and *ImportParameter*, are used to pass parameters between two pointcuts. They allow one to analyze a program's call graph in order to determine how to change function signatures for the passing of parameters associated with a given security hardening.

We first cast a glance at the shortcomings that were found during our previous experiments in Section 5.1. We introduce, in Section 5.2, some theoretical foundations allowing structural representation of programs. Afterwards, we dedicate Sections 5.3 and 5.4 to the matching pairs that are *GAFlow*/*GDFlow* and *ExportParameter*/*ImportParameter* in order to examine their usefulness and algorithms implementing them. We finally summarize our findings in Section 5.5.

5.1 Identified Shortcomings

As we have seen previously, some hardening activities cannot be implemented at all, or can only be implemented by increasing the software complexity. From those experiments we identified the following missing features in AOP languages:

Parameter Passing: It should be possible to pass parameter between advices in a manner that is simple, safe, and does not increase code complexity

Just in time library initialization: It should be possible to have the libraries used by the patterns initialized only when they are needed

Parameter name pointcut complement: It should be possible to complement pointcuts over function calls or execution with a selector based on the names of the variables passed.

The first two shortcomings are well illustrated by the Aspect excerpt in Listing 5.1, where we see the code injected around the `main` function and the use of a hash table to pass parameters. The functions of interest are underlined. During our research, we proposed solutions to two of the problems identified.

5.2 Program Representation

In order to propose the needed improvements to AOP languages, we must first obtain a familiarity with some concepts of program analysis, which emerged from the field of compilers and programming languages. Three instrumental concepts are lattices, control flow graphs, and call graphs.

5.2.1 Lattices

A lattice is a partially ordered set over which an ordering relation was applied [9] and it is typically represented as a directed acyclic graph, as illustrated in Figure 5.1. This ordering relation is reflexive, antisymmetric and transitive. A lattice is engineered in such a way that a greatest lower bound and least upper bound exist for each element of the set. The greatest

```
aspect SecureConnection {

 advice execution ("% main(...)") : around () {
   hardening_socketInfoStorageInit();
   hardening_initGnuTLSSubsystem(NONE);

       tjp->proceed();

       hardening_deinitGnuTLSSubsystem();
       hardening_socketInfoStorageDeinit();
 }

 advice call("% connect(...)") : around () {
   hardening_sockinfo_t socketInfo;
   const int cert_type_priority[3] = { GNUTLS_CRT_X509,
      GNUTLS_CRT_OPENPGP, 0};

   //initialize TLS session info
   gnutls_init (&socketInfo.session, GNUTLS_CLIENT);
   gnutls_set_default_priority (socketInfo.session);
   gnutls_certificate_type_set_priority (socketInfo.session,
      cert_type_priority);
   gnutls_certificate_allocate_credentials (&socketInfo.xcred);
   gnutls_credentials_set (socketInfo.session, GNUTLS_CRD_CERTIFICATE,
      socketInfo.xcred);

   //Connect
   tjp->proceed();
   if(*tjp->result()<0) {perror("cannot connect "); exit(1);}

   //TLS handshake
   socketInfo.isSecure = true;
   socketInfo.socketDescriptor = *(int *)tjp->arg(0);
   hardening_storeSocketInfo(*(int *)tjp->arg(0), socketInfo);
   gnutls_transport_set_ptr (socketInfo.session, (gnutls_transport_ptr)
       (*(int *)tjp->arg(0)));
   *tjp->result() = gnutls_handshake (socketInfo.session);
 }

   //replacing send() by gnutls_record_send() on a secured socket
 advice call("% send(...)") : around () {
   hardening_sockinfo_t socketInfo;
   socketInfo = hardening_getSocketInfo(*(int *)tjp->arg(0));

   if (socketInfo.isSecure)
      *(tjp->result())  = gnutls_record_send(socketInfo.session,
                          *(char**) tjp->arg(1), *(int *)tjp->arg(2));
   else
      tjp->proceed();
 }
};
```

Listing 5.1: Excerpt of an AspectC++ Aspect Hardening Connections Using GnuTLS

lower bound (or infimum) is the closest element that is smaller or equal to given elements in the set. The least upper bound (or supremum) is the closest element that is greater or equal to given elements in the set. A description of high-performance implementation of lattice operation is described in [4].

Lattices have been shown to be acceptable representation of programs for the purpose of programs' static analysis [18, 38], despite their limitations forbidding the representation of loops.

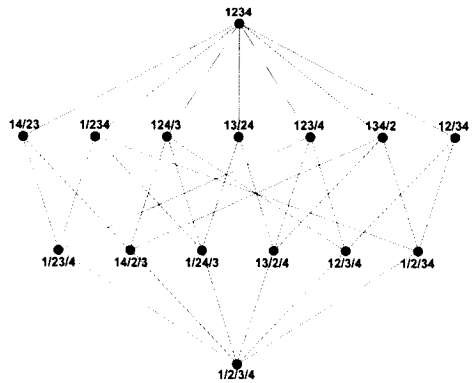


Figure 5.1: Lattice of Partitions of an Order 4 Set [67]

5.2.2 Control Flow Graphs

A control flow graph (CFG) is a potentially cyclic directed graph that is used to represent the program's possible flow of execution. Cycles are introduced in the graph by the use of loops. It is a form of program representation more appropriate than lattices thanks to its support for loops.

The graph is divided in *basic blocks*, which represents one or more code statements without branching. The edges of the graph are possible transitions from one basic block to another, typically due to a conditional branching (e.g. `if`) or a function call. It is

not possible to determine which path will be executed without the use of more powerful techniques such as data flow analysis.

Control flow graphs are used in optimizing compilers [1] as well as for certain static analysis methods, such as MOPS [61].

In Figure 5.2, we see the intraprocedural control flow graph of Java's `String.hashCode()` method, as generated by Würthinger's Hotspot Client Compiler Visualizer.

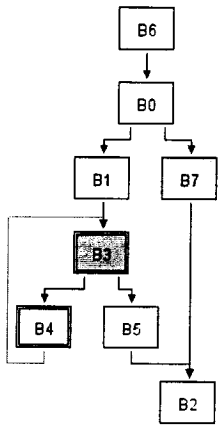


Figure 5.2: CFG of Java.Lang.String::hashCode

5.2.3 Call Graphs

A call graph (as illustrated in Figure 5.3) is a potentially cyclic directed graph that is used to represent the programs' calling structure. Cycles are introduced in the graph with the use of recursion.

Call graphs can be either context-insensitive or context sensitive. Context-insensitive call graph construction algorithms, such as proposed by Ryder [58], do not take in consideration the values of the variables used to call the functions. It is appropriate for procedural languages and yields a simple graph for which a function is represented only once in the

graph. Context sensitivity was proposed to deal with limitations to this approach found in more elaborate programming languages. Callahan et al. [15] proposed an algorithm dealing with procedural overloading based on the parameter types, and Grove et al. [27] also used the context-sensitive approach for call graph construction in object-oriented languages.

The reader interested in a sound overview on call graphs is referred to an elaborated study of different algorithms by Grove and Chambers [26].

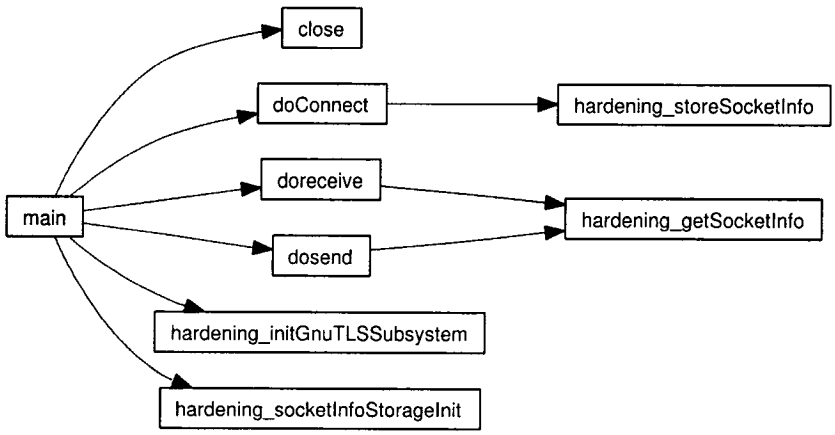


Figure 5.3: Call Graph From A Simple Program Generated by Doxygen [70]

5.2.4 Summary

In this short subsection, we quickly learnt about lattices, control flow graphs and call graphs. We saw that control flow graphs represent programs more closely than lattices, and that call graphs can be sensitive or insensitive to the context. These concepts will be useful for understanding the solutions we propose in this chapter.

5.3 *GAFlow* and *GDFlow*

The *GAFlow* and *GDFlow* pointcut designators [41, 42] have been developed in order to answer to the need of just-in-time library initialization. Their use allows aspect programmers to specify certain pointcuts as being relative to other pointcuts, thus avoiding having to specify them directly. This is very useful for the writing of security hardening patterns. For example, when some code injection simply needs to happen before or after another code injection, but where it is precisely injected does not matter. However, *GAFlow* and *GDFlow* are not only useful in the context of our approach, but also provide an enrichment to AOP that is worthwhile to examine by itself. In this section, we will thus define and examine these proposals in this manner, by examining the pointcut definition, their usefulness for security hardening purposes, and the algorithms that can be used to implement them.

5.3.1 Pointcut Definition

We provide in Figure 5.4 the syntax that defines a pointcut p that includes our proposal.

```
p ::= call(s) | execution(s) | gaflow(p) | gdflow(p)
                | p || p | p && p
```

Figure 5.4: *GAFlow* and *GDFlow* Extending AspectC++

In this representation, s is a function signature. As in AspectC++, pointcuts can be combined with the and (`&&`) and or (`||`) boolean operators. The *GAFlow* and the *GDFlow* are the new control flow based pointcut primitives. Their parameter is also a pointcut p. In the following, we present the definitions of each of pointcut.

These pointcut primitives operate on the CFG of a program. The input is a set of join points defined as a pointcut and the output is a single join point. In other words, if we are considering the CFG notations, the input is a set of nodes and the output is one node. The *GAFlow* output is (1) the closet common parent node of all the nodes specified in the input set (2) and through which all the possible paths that reach them pass. In the worst case, the closest common ancestor will be the entry block. The *GDFlow* output (1) is a common descendant of the selected nodes and (2) constitutes the first common node reached by all the possible paths emanating from the selected nodes. In the worst case, the first common descendant will be the exit block.

5.3.2 Usefulness of *GAFlow* and *GDFlow* for Security Hardening

Security hardening practices may require the injection of code around a set of join points or possible execution paths [7,30,62,73]. Examples of such cases would be the injection of security library initialization/deinitialization, privilege level changes, atomicity guarantee, logging, etc. The current AOP models only allow us to identify a set join points in the program, and therefore inject code before, after and/or around each one of them. However, to the best of our knowledge, none of the current pointcuts enable the identification of a join point common to a set of other join points where we can efficiently inject the code once for all of them. In the sequel, we present briefly examples of the usefulness of our proposed pointcuts for some security hardening activities.

Principle of Least Privilege: For processes implementing the principle of least privilege, it is necessary to increase the active rights before the execution of a sensitive operation,

and to relinquish such rights immediately after its completion. Our primitives can be used to deal with a group of operations requiring the same privilege by injecting the privilege adjustment code at the *GAFlow* and *GDFlow* join points. This is applicable only in the case where no unprivileged operations are in the execution path between the initialization and the deinitialization points.

Atomicity: In the case where a critical section may span across multiple program elements (such as function calls), there is a need to enforce mutual exclusion using tools such as semaphores around the critical section. The beginning and end of the critical section can be targeted using the *GAFlow* and *GDFlow* join points.

Logging: It is possible that a set of operations are of interest for logging purposes, but that their individual log entry would be redundant or of little use. This is why it is desirable to use *GAFlow* and/or *GDFlow* in order to insert log statements before or after a set of interesting transactions.

Security Library Initialization/Deinitialization: In the case of security library initialization (e.g. access control, authorization, cryptography), our primitives allow us to initialize and deinitialize the needed library only for the branches of code where they are needed by identifying their *GAFlow/GDFlow*. Having both primitives would also avoid the need to keep global state variables about the current state of library initialization. Let us consider the case of securing a connection. With the current AOP pointcuts, as shown in Listing 5.1 the aspect targets the `main` as the location for the TLS library initialization and deinitialization. Another possible solution could be the loading and unloading of the library before and after its use, which may cause runtime problems since API-specific data structures could

be needed for other functions. However, in the case of large or embedded applications the two solutions create an accumulation of code injection statements that would create a significant, and possibly useless, waste of system resources. In Listing 5.2, we provide an aspect implemented using the proposed pointcut, instead of an `advice` that is executed `around` the function `main`. The resulting code will be more efficient and applicable on a wider range of application.

5.3.3 General Advantages of *GAFlow* and *GDFlow*

It is clear that our proposed primitives support the principle of separation of concerns by allowing the implementation of program modifications on sets of join points based on a specific concern (as previously exemplified). We further upon this benefit by presenting some general advantages of our proposed pointcuts:

Ease of Use and Higher Abstraction: Programmers can target places in the application's control flow graph where to inject code before or after a set of join points without needing to manually determine the precise point where to do so.

Ease of Maintenance: Programmers can change the program structure without needing to rewrite the associated aspects that were relying on explicit knowledge of the structure in order to pinpoint where the advice code would be injected. For example, if we need to change the execution path to a particular function (e.g. when performing refactoring), we also need to manually find the new common ancestor and/or descendant, whereas this would be done automatically using our proposed pointcuts.

Optimization: Programmers can inject certain pre-operations and post-operations only where needed in the program, without having to resort to injection in the catch-all `main`. This is replaced by executing them only once around the *GAFlow* and *GDFlow*.

5.3.4 Algorithms and Implementation

In this section, we present the elaborated algorithms for graph labeling, *GAFlow* and *GDFlow*. We chose to use the programs' CFG in order to implement *GAFlow* and *GDFlow* because of shortcomings with lattice theory. Lattice theory defines the infimum and supremum operations that are somewhat similar to *GAFlow* and *GDFlow*. However, their results do not guarantee that all paths will be traversed by the results of these operations, which is a central requirement for *GAFlow* and *GDFlow*. Moreover, the lattices do not correctly represent all kinds of programs, due to their inability to support loops.

Algorithms that operate on graphs have been developed for decades now, and many graph operations (such as finding ancestors, finding descendants, finding paths and so on) are considered to be common knowledge in computer science. Despite this theoretical richness, we are not aware of existing methods allowing us to efficiently determine the *GAFlow* and *GDFlow* for a particular set of join points in a CFG by considering all the possible paths.

We assume that our CFG is shaped in the traditional form, with a single start node (entry block) and a single end node (exit block). In the case of program with multiple starting points, we consider each starting point as a different program in our analysis. In the case of multiple ending points, we consider them connected to a single end point. With

these assumptions in place, we ensure that our algorithms will return a result (in the worst case, the entry or exit block) and that this result will be a single and unique node for all inputs. We implemented these algorithms to operate on arbitrary graphs and experimentally explored their correctness.

Graph Labeling: In order to determine the *GAFlow* and *GDFlow*, we choose to use a graph labeling algorithm adapted from an algorithm developed by our colleagues Soeanu and Jarraya [16] in the Computer Security Laboratory. Algorithm 5.1 describes our graph labeling method. Since our graph may contain loops, we modified the algorithm in a way that a node will not be labeled twice if the parent is one of its descendants. In such a situation, the recursion is stopped thus guaranteeing termination of the algorithm. This approach facilitates processing, since the hierarchy is encoded in the label set associated to each node. As we will see later, *GAFlow* and *GDFlow* algorithms are dependent on this labeling to facilitate processing.

Each node down the hierarchy is labeled in the same manner as the table of contents of a book (e.g. 1., 1.1., 1.2., 1.2.1., ...), as depicted by Algorithm 5.1, where the operator $+_c$ denotes string concatenation (with implicit operand type conversion). To that effect, the labeling is done by executing algorithm 5.1 on the *entry* node with label "0.", thus recursively labeling all nodes. An example is shown in Figure 5.5.

GAFlow*:* In order to compute the *GAFlow*, we developed a mechanism that operates on the labeled graph. We compare all the hierarchical labels of the selected nodes in the input set and find the largest common prefix they share. The node labeled with this largest common prefix is the closest guaranteed ancestor. We insured that the *GAFlow* result is

Algorithm 5.1 Hierarchical Graph Labeling Algorithm

```
1: function labelNode(Node s, Label l):
2: s.labels ← s.labels ∪ {l}
3: NodeSequence children = s.children()
4: for k = 0 to |children| − 1 do
5:    child ← children[k]
6:    if ¬hasProperPrefix(child, s.labels) then
7:       labelNode(child, l +c k +c ".");
8:    end if
9: end for
10:
11: function hasProperPrefix(Node s, LabelSet parentLabels):
12: if s.label = ϵ then
13:    return false
14: end if
15: if ∃s ∈ Prefixes(s.label) : s ∈ parentLabels then
16:    return true
17: else
18:    return false
19: end if
20:
21: function Prefixes(Label l):
22: LabelSet labels ← ∅
23: Label current ←""
24: for i ← 0 to l.length() do
25:    current.append(l.charAt(i))
26:    if Label1.charAt(i) = '.' then
27:       labels.add(current.clone())
28:    end if
29: end for
```

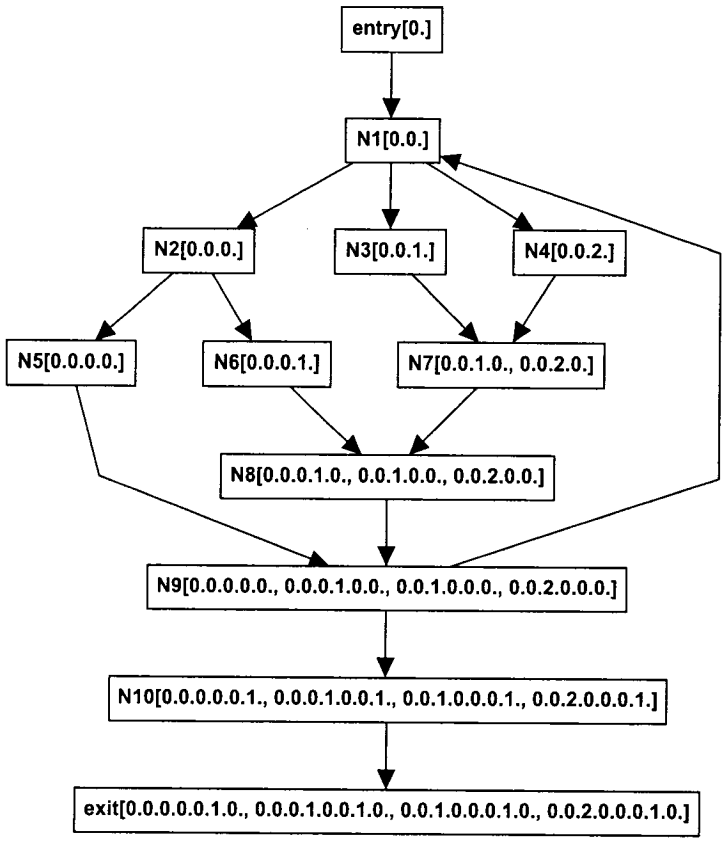


Figure 5.5: Sample Labeled Graph

a node through which all the paths that reach the selected nodes pass by considering all the labels of each node. This is elaborated in Algorithm 5.2. Figure 5.6 shows a graphical representation of *GAFlow*.

GDFlow: The closest guaranteed descendant is determined by elaborating a mechanism that operates on a labeled CFG of a program. By using Algorithm 5.3, we obtain the sorted list of all the common descendants of the selected nodes in the input list of the pointcut. The principle of this algorithm is to calculate the set of descendants of each of the input nodes and then perform the intersection operation on them. The resulting set contains the common descendants of all the input nodes. Then, we sorted them based on their path length.

Algorithm 5.2 Algorithm to determine *GAFlow*

Require: $SelectedNodes$ is initialized with the contents of the pointcut match
Require: $Graph$ has all its nodes labeled

```
1: function gaflow(NodeSet SelectedNodes):
2: LabelSequence Labels ← ∅
3: for all node ∈ SelectedNodes do
4:    Labels ← Labels ∪ node.labels()
5: end for
6: return GetNodeByLabel(FindCommonPrefix(Labels))
7:
8: function FindCommonPrefix (LabelSequence Labels):
9: if |Labels| = 0 then
10:    return error
11: else if |Labels| = 1 then
12:    return Labels.removeHead()
13: else
14:    Label Label1 ← Labels.removeHead()
15:    Label Label2 ← Labels.removeHead()
16:    if |Labels| = 2 then
17:       for i ← 0 to min(Label.length(), Label2.length() do
18:          if Label1.charAt(i) ≠ Label2.charAt(i) then
19:             return Label1.substring(0, i − 1)
20:          end if
21:       end for
22:       return Label1.substring(0, min(Label.length(), Label2.length())
23:    else
24:       Label PartialSolution ← FindCommonPrefix(Label1, Label2)
25:       Labels.append(PartialSolution)
26:       return FindCommonPrefix(Labels)
27:    end if
28: end if
```

Algorithm 5.3 Algorithm to Determine the Common Descendants

Require: $SelectedNodes$ is initialized with the contents of the pointcut match
Require: $Graph$ has all its nodes labeled

```
1: function findCommonDescendants(NodeSet SelectedNodes):
2: NodeSet PossibleSolutions ← Graph.allNodes()
3: for all node ∈ SelectedNodes do
4:    PossibleSolutions ← PossibleSolutions ∩ node.AllDescendants()
5: end for
6: Create OrderedSolutions by sorting PossibleSolutions by increasing path length
   between the solution and the nodes in SelectedNodes
7: return OrderedSolutions
```

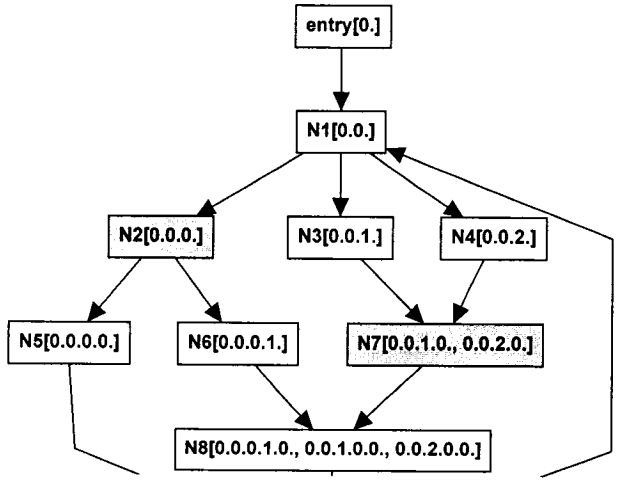


Figure 5.6: *GAFlow* for N2 and N7

Algorithm 5.4 determines the closest guaranteed descendant. It takes first the result of Algorithm 5.3, which is considered as its list of possible solutions. Then, it iterates through the ordered list and examines the current node against the previously retained solution, which we call the candidate. For each selected node, we count the number of labels of the candidate that have proper prefixes identical to the labels of the considered selected node. The resulting candidate of the first iteration is the first encountered node with the largest label count. This candidate is the starting one of the next iteration and so on until all the selected nodes are examined. The final candidate of the last iteration is returned by the algorithm as the closest guaranteed descendant. This approach is much more complex than what we used for *GAFlow*. It was however justified by the need to handle the case of loops, which the *GAFlow* does not need to consider in its treatment. The Figure 5.7 shows a graphical representation of *GDFlow*.

Algorithm 5.4 Algorithm to Determine *GDFlow*

Require: $SelectedNodes$ is initialized with the contents of the pointcut match
Require: *Graph* has all its nodes labeled

```
1: function gdflow(NodeSet SelectedNodes):
2: NodeSequence PossibleSolutions ← findCommonDescendants(SelectedNodes)
3: IntCandidateIndex ← 0
4: for all node ∈ SelectedNodes do
5:    CandidateIndex ← findBestCandidate(PossibleSolutions, CandidateIndex, node)
6: end for
7: return PossibleSolutions[Candidate]
8:
9: function findBestCandidate(NodeSequence possibleSolutions, int CandidateIndex,
   Node selectedNode)
10: PreviousFoundPrefixes ← 0
11: for i ← CandidateIndex to |possibleSolutions| − 1 do
12:    Int sol ← possibleSolutions[i]
13:    Int foundPrefixes ← countProperPrefixes(sol, node)
14:    if (PreviousFoundPrefixes < foundPrefixes) ∨ ∃child ∈ sol.children() :
       hasProperPrefix(sol, child.labels()) then
15:       CandidateIndex ← i
16:    end if
17: end for
18: return CandidateIndex
19:
20: function countProperPrefixes(Node candidate, Node selectedNode):
21: Int count ← 0
22: for all candidateLabel ∈ candidate.labels() do
23:    for all selectedNodeLabel ∈ selectedNode.labels() do
24:       if ∃p ∈ Prefixes(candidateLabel) : p = selectedNodeLabel then
25:          count + +
26:       end if
27:    end for
28: end for
29: return count
```

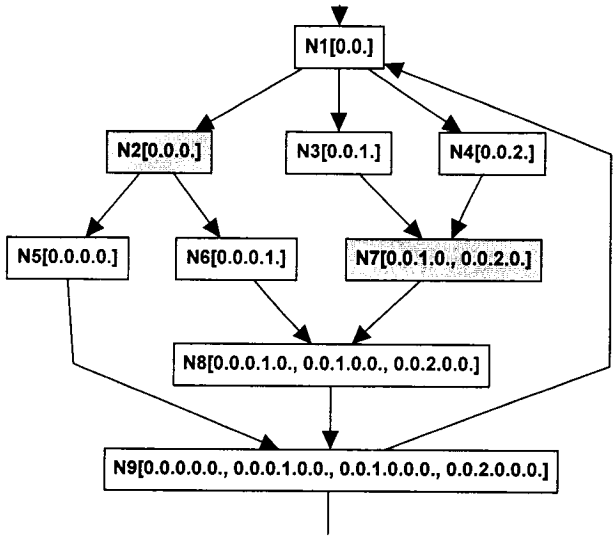


Figure 5.7: *GDFlow* for N2 and N7

5.4 Parameter Passing

In this section, we present the *ExportParameter* and *ImportParameter* primitives required for parameter passing [49]. Those primitives allow the transfer of parameters between two advices via the intermediary procedures' parameter list. Moreover, we illustrate their usefulness for security hardening and their algorithms. We define their syntax, derived from AspectC++, as an optional program transformation section in an advice declaration as illustrated in Figure 5.8, where e and i are respectively the new *ExportParameter* and *ImportParameter*. The arguments of *ExportParameter* are the parameters to pass, while the arguments of *ImportParameter* are the parameters to receive. The two primitives are meant to be used together in order to provide the information needed for parameter passing from one join point to another.

Our proposed approach for parameter passing operates on the context-insensitive call graph of a program [26], with each node representing a function and each arrow representing a call site. It exports the parameter over all the possible paths going from the origin to

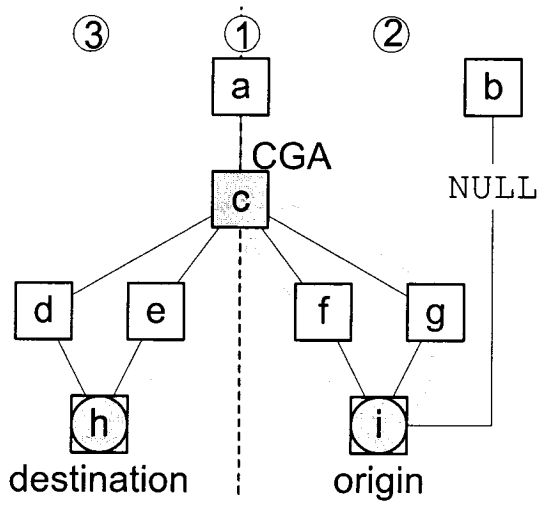
```
parameter ::= <type> <identifier>
paramList ::= parameter [,paramList]
e ::= exportParameter(<paramList>)
i::= importParameter(<paramList>)
advice <target-pointcut> : (before|after|around) (<arguments>)
[: e | i | e,i] {<advice-body>}
```

Figure 5.8: Syntax for Parameter Passing Primitives Extending Aspect C++

the destination nodes. This is achieved by performing the following three steps: (1) Calculating the closest guaranteed ancestor (CGA, using the *GAFlow* algorithm) of the origin and destination join points, (2) passing the parameter from the origin to the aforementioned node and then (3) from this node to the destination. Figure 5.9 illustrates our approach. For instance, to pass the parameter from h to g, their *GAFlow*, which is b in the this case, is first identified. Afterwards, the parameter is passed over all the paths from h to b, then from b to g again over all the paths.

By passing the parameter from the origin to *GAFlow* and then to the destination, we ensure that the parameter will be effectively passed through all the possible execution paths between the two join points. Otherwise, the parameter could not be passed or passed without initialization, which would create software errors and affect the correctness of the solution.

The parameter passing capability that we propose changes the function signatures and the relevant call sites in order to propagate useful security hardening variables via `inout` function parameters. It changes the signatures of all the functions involved in the call graph between the exporting and importing join points. All calls to these functions are modified to pass the parameter as is, in the case of the functions involved in this transmission path (e.g. nodes b, c, d, e, and f).



1: Identify gaflow 2: Origin → gaflow 3: gaflow → destination

Figure 5.9: Parameter Passing in a Call Graph

5.4.1 Usefulness of *ImportParameter* and *ExportParameter* for Security Hardening

It is often necessary to pass state information from one part of the program to another in order to perform security hardening. For instance, in the example provided in Listing 5.1, we need to pass the `gnutls_session_t` data structure from the advice around `connect` to the advice around `send`, `receive` and/or `close` in order to properly harden the connection. The current AOP models do not allow us to perform such operations without the need for relatively complex data structures and algorithms.

In Listing 5.2, we use our proposed approach for parameter passing. All data structures and algorithms (underlined in Listing 5.1) are removed and replaced by the primitives for exporting and importing. An `exportParameter` for the parameters `session` and `xcred` is added on the declaration of the advice of the pointcut that identifies the function

`connect`. Moreover, an `importParameter` for the parameter `session` is added on the declaration of the advice of the pointcut that identifies the function `send`.

5.4.2 Algorithms and Implementation

The elaborated algorithms for the implementation of parameter passing operate on a program's call graph. The origin node is the pointcut where the *ExportParameter* is called, while the destination node is the pointcut where *ImportParameter* is called.

After calculating the closest guaranteed ancestor of the two pointcuts specified by the two primitives, Algorithm 5.6 is performed first to pass the parameter from the origin to the closest guaranteed ancestor, then executed another time to pass the parameter from the closest guaranteed ancestor to the destination. This procedure is described in Algorithm 5.5 and operates on a per-parameter basis.

Algorithm 5.5 Algorithm to Pass the Parameter between two pointcuts

```
1: function passParameter(Node origin, Node end,Parameter param):
2: if origin = destination then
3:     return success
4: end if
5: start ← ClosestGuaranteedAncestor(origin, end)
6: passParamOnBranch(start, origin, param)
7: node.addLocalVariable(param)
8: passParamOnBranch(start, end, param)
```

Algorithm 5.6 is a building block that allows the modification of function signatures and calls in a way that would keep the program's syntactical correctness and intent, meaning that it would still compile and behave the same. It finds all the paths between an origin node and a destination node in a call graph. For each path, it propagates the parameter from the called function to the callee, starting from the end of the path. In order to be optimal, it

```
aspect SecureConnection {

 advice gaflow(call("% connect(...)") || call("% send(...)") || call("%
     recv(...)")): before(){
   gnutls_global_init();
 }

 advice gdflow(call("% connect(...)") || call("% send(...)") || call("%
     recv(...)") || call("% close(...)")): after(){
   gnutls_global_deinit();
 }

 advice call("% connect(...)") : around () : exportParameter(
    gnutls_session session, gnutls_certificate_credentials xcred){
   //variables declared
   static const int cert_type_priority[3] = { GNUTLS_CRT_X509,
      GNUTLS_CRT_OPENPGP, 0};

   //initialize TLS session info
   gnutls_init (&session, GNUTLS_CLIENT); gnutls_set_default_priority (
      session);
   gnutls_certificate_type_set_priority (session, cert_type_priority);
      gnutls_certificate_allocate_credentials (&xcred);
   gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);

   //Connect
   tjp->proceed();
   if(*tjp->result()<0) {perror("cannot connect "); exit(1);}

   //TLS handshake
   gnutls_transport_set_ptr (session, (gnutls_transport_ptr) (*(int *)
      tjp->arg(0)));
   *tjp->result() = gnutls_handshake (session);
 }

 advice call("% send(...)") : around () : importParameter(
    gnutls_session session){
   if (session != NULL) //if the channel is secured, replace the send
      by gnutls_send
      *(tjp->result())  = gnutls_record_send(*session, *(char**) tjp->
         arg(1), *(int *)tjp->arg(2));
   else
      tjp->proceed();
 }
};
```

Listing 5.2: Hardening Aspect for Securing Connections using *ImportParameter*, *ExportParameter*, *GAFlow* and *GDFlow*

modifies all the callers only one time and keeps track of the modified nodes. When a caller is not involved in the transmission path, only the call is modified, and a null value is passed to the called function.

Algorithm 5.6 Algorithm to Pass a Parameter Between Two Nodes of a Call Graph

```
1: function passParamOnBranch(Node origin, Node destination, Parameter param):
2: if origin = destination then
3:    return success
4: end if
5: PathSequence paths ← findPathsBetween(origin, destination)
6: for all path ∈ paths do
7:    path.remove(origin)
8:    while ¬path.isEmpty() do
9:       currnode ← path.tail()
10:      path.remove(currnode)
11:      if ¬node.signature.isModified() ∧
         ¬∃parameter ∈ currnode.signature() : parameter = param then
12:         node.signature.addParameter(param)
13:         node.signature.markModified()
14:         modifyFunctionsCallsTo
            (currNode, param)
15:      end if
16:   end while
17: end for
18: return success
19:
20: function modifyFunctionsCallsTo(Node currnode, Parameter param):
21: for all caller ∈ currnode.getCallers() do
22:    for all call ∈ caller.getCallsTo(node) : ¬call.modified do
23:       call.parameters.add(param)
24:       call.modified = true
25:    end for
26: end for
```

5.5 Summary

We presented in this chapter the shortcomings of AOP for many security hardening concerns and proposed improvements to the current AOP technologies by proposing new pointcuts and primitives. The two proposed pointcuts, *GAFlow* and *GDFlow*, allow one to identify particular join points in a program's CFG. They allow us to locate, in a maintenance-friendly manner, where to inject code for different security hardening activities, such as security library initialization and deinitialization. The two proposed primitives to AOP weaving capabilities, *ExportParameter* and *ImportParameter*, allow parameter passing from one advice to the another through a program's context-insensitive call graph and offer a lightweight solution to the problem of inter-advice parameter passing. We have shown algorithms that operate on procedural programs' CFG and call graph to demonstrate the implementability of the proposals.

Chapter 6

Case Study: Adding TLS Support to APT

A first issue is the securing of channels between two communicating parties to avoid eavesdropping, tampering with the transmission or session hijacking. In this section, we illustrate our elaborated solutions for securing the connections of client applications by following the approach's methodology and using the proposed *SHL* language. We added HTTPS support to APT, an automated package downloader and manager for the Debian Linux distribution. It is written in C++ and is composed of more than 23,000 source lines of code (statistics generated using David A. Wheeler's 'SLOCCount' and APT version 0.5.28). It obtains packages via local file storage, FTP, HTTP, etc.

In the sequel, we will briefly present the architecture of APT in Section 6.1. Then, we will see the hardening plan we specified in Section 6.2 before examining the hardening pattern used in Section 6.3. Afterwards, in Section 6.4, we see an AspectC++ aspect that corresponds to the refining step of our approach. We also see non-aspectual refining results

done manually on the application in Section 6.5. We then see the results of our case study in Section 6.6. Finally, in Section 6.7, we summarize the findings of this chapter.

6.1 Architecture of APT

APT is organized in a few simple components that allow extensibility. All package acquisition methods are separated from the package management logic and are grouped individually as programs. The library (`libapt`) creates the process of the method and communicates with it using the created process' standard input and output. The library sends acquisition requests to the method, which will parse and process it. The acquisition method is responsible for writing the downloaded file to disk. The functions of the library can be used by different software packages, but the source code includes many command-line tools. In our case, we used the `apt-get` command-line tool and we created an HTTPS method based on the existing HTTP method. These explanations are visually summarized in Figure 6.1.

6.2 Hardening Plan

In Listing 6.1, we include an example of an effective security hardening plan for securing the connection of the HTTP method. It specifies to add hardening on two files: `http.cc` and `connect.cc`. The language to use is `C` (since APT is written in `C++`), and the library used will be GnuTLS, using SSL in client mode, with default settings. The plan, written in *SHL*, is very expressive and yet simple to read.

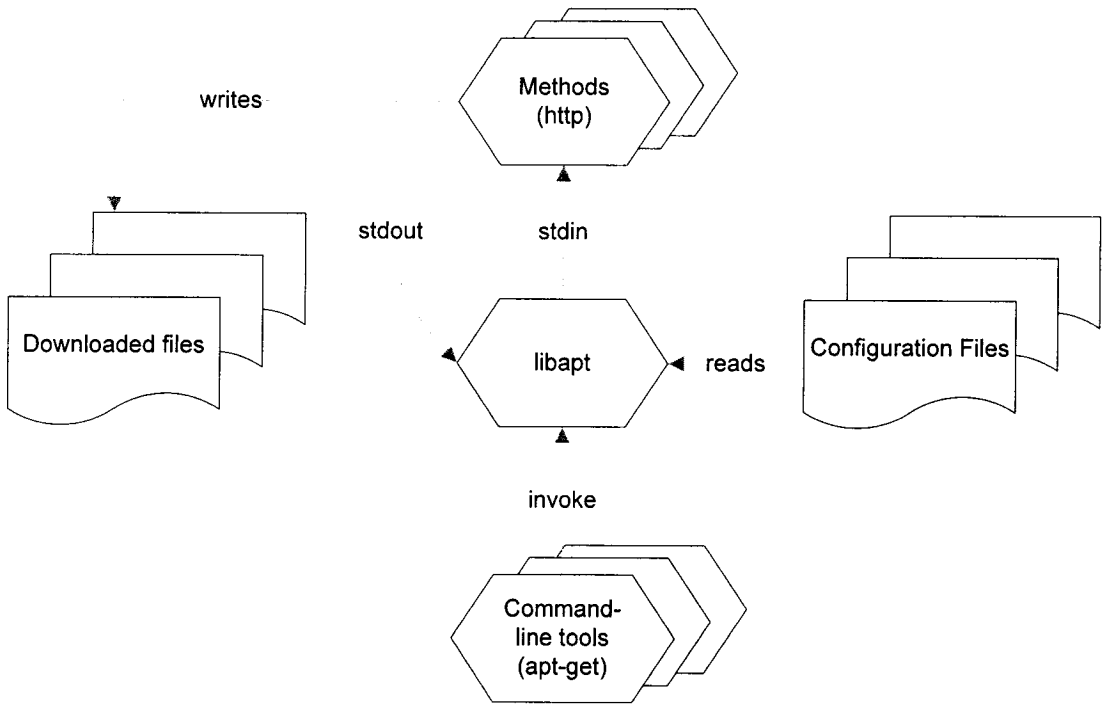


Figure 6.1: High-Level Architecture of APT

6.3 Hardening Pattern

Listings 6.2 and 6.3 show the practical pattern specification for securing the connection of the aforementioned application using GnuTLS. We implemented it without certificate verification for the sake of simplicity, and forced the socket to be non-blocking for the connection and handshake in order to avoid connection issues.

```
Plan      APT_Secure_Connection_Plan
BeginPlan
    PatternName     Secure_Connection_Pattern
    Parameters
        Language=C
        API=GNUTLS
        Peer=Client
    Where     http.cc connect.cc
EndPlan
```

Listing 6.1: Hardening Plan for Adding HTTPS to APT

6.4 Manual Refinement Generating an Aspect

We refined, using AspectC++, the aspect corresponding to the pattern presented in Listing 6.2, an excerpt of which is shown in Listings 6.4 and 6.5. In the Listing, we underlined the usage of our security hardening APIs, written for this pattern. We re-used the hash table approach that we described in Chapter 4 in order to perform parameter passing.

6.5 Manual Refinement Modifying the Source Code

We also used a manual approach to refinement, where the source code is modified in compliance with the pattern in Listing 6.2. The results, showed in unified diff format [44], are in Listing 6.6, Listing 6.7 and Listing 6.8. The files were edited for conciseness.

6.6 Experimental Results

In order to verify the hardened application, we used the Debian apache-ssl package, an HTTP server that accepts only Secure Socket Layer (SSL) and TLS connections. We populated the server with a software repository compliant with APT's requirements, so that APT

```
Pattern Secure_Connection_Pattern
Parameters
      Language=C
      API=GNUTLS
      Peer=Client

BeginPattern

Before
GAFlow<FunctionCall <send> || FunctionCall <recv >|| FunctionCall <
   connect> || FunctionCall <close> >
BeginBehavior
  gnutls_global_init();
EndBehavior

Before
FunctionCall <connect> (socket,*,*)
ExportParameter <xcred>
ExportParameter <session>
ExportParameter <socketflags>
BeginBehavior
  const int cert_type_priority[3] = { GNUTLS_CRT_X509,
     GNUTLS_CRT_OPENPGP, 0};
  gnutls_init (session, GNUTLS_CLIENT);
  gnutls_set_default_priority (session);
  gnutls_certificate_type_set_priority (session, cert_type_priority);
  gnutls_certificate_allocate_credentials(xcred);
  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);
  int socketflags = fcntl(socket,F_GETFL);
  if ((socketflags & O_NONBLOCK) != 0) fcntl(socket,F_SETFL, socketflags
      ^ O_NONBLOCK);
EndBehavior

After
FunctionCall <connect> (socket,*,*)
ImportParameter <session>
ImportParameter <socketflags>
BeginBehavior
  gnutls_transport_set_ptr(session, socket);
  gnutls_handshake (session);
  if ((socketflags & O_NONBLOCK) != 0){
    fcntl(socket,F_SETFL, socketflags); //restore non-blocking state if
       it was like that
    gnutls_transport_set_lowat(session,0); //now make gnutls aware that
       we are dealing with non-blocking sockets
  }
EndBehavior
```

Listing 6.2: Hardening Pattern for Securing Connection (part 1)

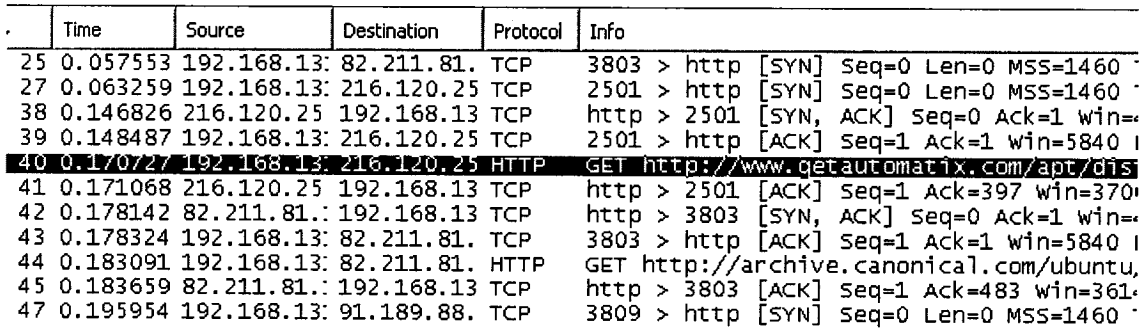
```
Replace
FunctionCall <send> (*,data,datalen)
ImportParameter <session>
BeginBehavior
  gnutls_record_send(session, data, datalen);
EndBehavior

Replace
FunctionCall <recv> (*,data,datalen)
ImportParameter <session>
BeginBehavior
  gnutls_record_recv(session, data, datalen);
EndBehavior

Before
FunctionCall <close>
ImportParameter <xcred>
ImportParameter <session>
BeginBehavior
  gnutls_bye(session, GNUTLS_SHUT_RDWR);
  gnutls_deinit(session);
  gnutls_certificate_free_credentials(xcred);
EndBehavior


After
GDFlow<FunctionCall <send> || FunctionCall <recv>|| FunctionCall <
   connect> || FunctionCall <close> >
BeginBehavior
  gnutls_global_deinit();
EndBehavior

EndPattern
```

Listing 6.3: Hardening Pattern for Securing Connection (part 2)

can connect automatically to the server and download the needed metadata in the repository. The resulting hardened software was capable of performing both HTTP and HTTPS package acquisition, based on the parameters in the configuration file. After deploying our modified package, we tested successfully its functionality by refreshing APT's package database, which forced the software to connect to both our local web server (Apache-ssl) using HTTPS and remote servers using HTTP to update its list of packages. The experimental results in Figures 6.2 and 6.3 and Listing 6.4 show that the new secured application is able to connect using both HTTP and HTTPS connections.

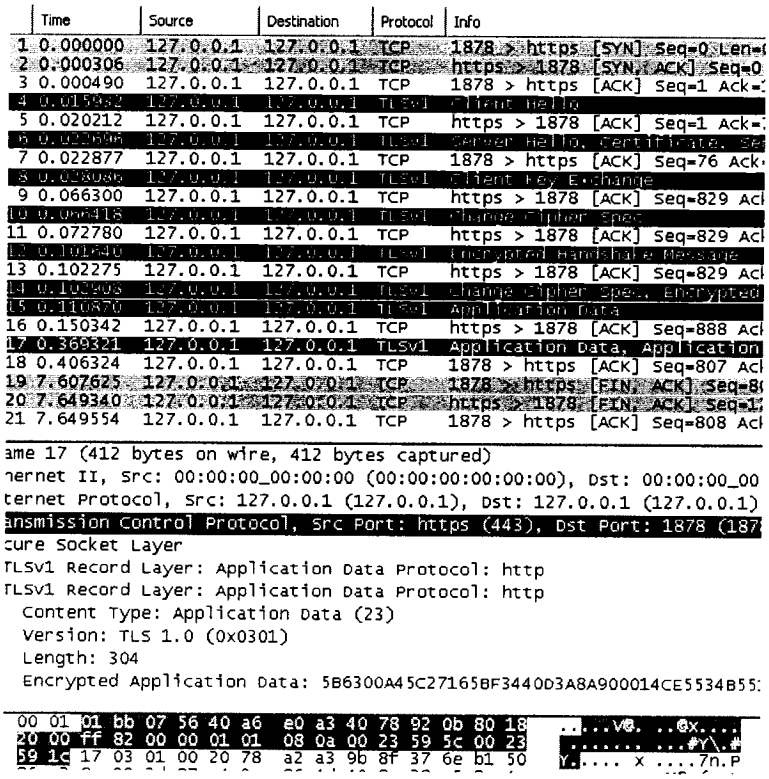
We now provide brief explanations of our results. Figure 6.2 shows the packet capture, obtained using WireShark [39], of the unencrypted HTTP traffic between our version of APT and its remote package repositories, configured to use a non-secure connection. The highlighted line shows an HTTP connection to the `www.getautomatix.com` APT package repository. On the other hand, in Figure 6.3, we see that the connections between our version of APT and the local web server are using HTTPS. Specifically, the highlighted line shows TLSv1 application data exchanged in encrypted form. Moreover, we see clearly that all the information exchanged was encrypted and passed via TLS/SSL. Moreover, Figure 6.4 shows an extract of Apache's access log, edited here for conciseness, where the package metadata was successfully obtained from our local server by the applications.

We must also mention the ease of use for the AOP-based approach. It required us only a few work days to examine the code, find the injection points, and adjust the pattern to fit our needs. The solution was modular, clean and effective, and would have been much shorter had we been more experienced using the Debian build system. On the other hand, it took us a few more days to add the HTTPS functionality. That solution was, from a



	Time	Source	Destination	Protocol	Info
25	0.057553	192.168.13:	82.211.81.	TCP	3803 > http [SYN] Seq=0 Len=0 MSS=1460
27	0.063259	192.168.13:	216.120.25	TCP	2501 > http [SYN] Seq=0 Len=0 MSS=1460
38	0.146826	216.120.25	192.168.13	TCP	http > 2501 [SYN, ACK] Seq=0 Ack=1 Win=
39	0.148487	192.168.13:	216.120.25	TCP	2501 > http [ACK] Seq=1 Ack=1 Win=5840
40	0.170727	192.168.13.	216.120.25	HTTP	GET http://www.getautomatix.com/apt/dis
41	0.171068	216.120.25	192.168.13	TCP	http > 2501 [ACK] Seq=1 Ack=397 Win=370
42	0.178142	82.211.81.:	192.168.13	TCP	http > 3803 [SYN, ACK] Seq=0 Ack=1 Win=
43	0.178324	192.168.13:	82.211.81.	TCP	3803 > http [ACK] Seq=1 Ack=1 Win=5840
44	0.183091	192.168.13:	82.211.81.	HTTP	GET http://archive.canonical.com/ubuntu,
45	0.183659	82.211.81.:	192.168.13	TCP	http > 3803 [ACK] Seq=1 Ack=483 Win=361
47	0.195954	192.168.13:	91.189.88.	TCP	3809 > http [SYN] Seq=0 Len=0 MSS=1460

Figure 6.2: Packet Capture of Unencrypted APT Traffic



	Time	Source	Destination	Protocol	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	1878 > https [SYN] Seq=0 Len=
2	0.000306	127.0.0.1	127.0.0.1	TCP	https > 1878 [SYN, ACK] Seq=0
3	0.000490	127.0.0.1	127.0.0.1	TCP	1878 > https [ACK] Seq=1 Ack=
4	0.015952	127.0.0.1	127.0.0.1	TLSv1	Client Hello
5	0.020212	127.0.0.1	127.0.0.1	TCP	https > 1878 [ACK] Seq=1 Ack=
6	0.022606	127.0.0.1	127.0.0.1	TLSv1	Server Hello, Certificate, Se
7	0.022877	127.0.0.1	127.0.0.1	TCP	1878 > https [ACK] Seq=76 Ack=
8	0.028086	127.0.0.1	127.0.0.1	TLSv1	Client Key Exchange
9	0.066300	127.0.0.1	127.0.0.1	TCP	https > 1878 [ACK] Seq=829 Ac
10	0.066418	127.0.0.1	127.0.0.1	TLSv1	Change Cipher Spec
11	0.072780	127.0.0.1	127.0.0.1	TCP	https > 1878 [ACK] Seq=829 Ac
12	0.101640	127.0.0.1	127.0.0.1	TLSv1	Encrypted Handshake Message
13	0.102275	127.0.0.1	127.0.0.1	TCP	https > 1878 [ACK] Seq=829 Ac
14	0.102906	127.0.0.1	127.0.0.1	TLSv1	Change Cipher Spec, Encrypted
15	0.110870	127.0.0.1	127.0.0.1	TLSv1	Application Data
16	0.150342	127.0.0.1	127.0.0.1	TCP	https > 1878 [ACK] Seq=888 Ac
17	0.369321	127.0.0.1	127.0.0.1	TLSv1	Application Data, Application
18	0.406324	127.0.0.1	127.0.0.1	TCP	1878 > https [ACK] Seq=807 Ac
19	7.607625	127.0.0.1	127.0.0.1	TCP	1878 > https [FIN, ACK] Seq=8
20	7.649340	127.0.0.1	127.0.0.1	TCP	https > 1878 [FIN, ACK] Seq=1
21	7.649554	127.0.0.1	127.0.0.1	TCP	1878 > https [ACK] Seq=808 Ac

```
ame 17 (412 bytes on wire, 412 bytes captured)
nernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00
ternet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
ansmission Control Protocol, Src Port: https (443), Dst Port: 1878 (187
cure Socket Layer
TLSv1 Record Layer: Application Data Protocol: http
TLSv1 Record Layer: Application Data Protocol: http
  Content Type: Application Data (23)
  Version: TLS 1.0 (0x0301)
  Length: 304
  Encrypted Application Data: 5B6300A45C27165BF3440D3A8A900014CE5534B55
```

```
00 01 01 bb 07 56 40 a6  e0 a3 40 78 92 0b 80 18   .....V@. ..@x....
20 00 ff 82 00 00 01 01  08 0a 00 23 59 5c 00 23   ........ ...#Y\.#
59 1c 17 03 01 00 20 78  a2 a3 9b 8f 37 6e b1 50   Y..... x ....7n.P
```

Figure 6.3: Packet Capture of SSL-protected APT Traffic

```
127.0.0.1 - - [04/Apr/2007:11:52:02 -0400] "GET https2://localhost:443/
   apt/dists/dapper/main/binary-i386/Release HTTP/1.1" 304 - "-" "
   Debian APT-HTTP/1.3" "-"
127.0.0.1 - - [04/Apr/2007:13:53:17 -0400] "GET https2://localhost:443/
   apt/dists/dapper/main/binary-i386/Packages.gz HTTP/1.1" 304 - "-" "
   Debian APT-HTTP/1.3" "-"
127.0.0.1 - - [04/Apr/2007:13:53:17 -0400] "GET https2://localhost:443/
   apt/dists/dapper/main/binary-i386/Release HTTP/1.1" 304 - "-" "
   Debian APT-HTTP/1.3" "-"
```

Figure 6.4: Excerpt of Apache Access Log

software quality point of view, very poor, and it would have needed further digging into the code and heavy refactoring in order to offer a clean solution. This experiment suggests the AOP-based solution is faster, in terms of hours of work, by approximately an order of magnitude, although more strict experimentation is needed in order to bring conclusive results on this matter.

6.7 Summary

In this chapter, we saw the architecture of APT, an automated package downloader and manager for Debian. We wrote an hardening plan to add HTTPS functionality to the program, and we used a pattern that supplied this functionality. The experimental results presented together with the security hardening plans and patterns, which are elaborated using *SHL*, explore the efficiency and relevance of our proposition. We see that the manual refinement can result in both an aspect or directly modified code that share the same functional result. More interesting results should come when AOP tools will support the full range of primitives integrated in *SHL*, in which case lightweight aspects will be generated and that no manual transformation step will be required. Our approach was shown to be very interesting, as it lowered the level of digging necessary in order to perform the hardening, lowered the time needed to do so, and that the solution was easily expandable to other hardenings if desired.

```
aspect https{

advice execution( "% HttpMethod::Loop()" ) : around () {
//init gnutls lib
hardening_initGnuTLSSubsystem(NONE); hardening_socketInfoStorageInit();
tjp->proceed();
//deinit libs
hardening_socketInfoStorageDeinit(); hardening_deinitGnuTLSSubsystem();
}
advice call("% connect(...)") : around () {
    //variables declared
    hardening_sockinfo_t socketInfo;
    const int cert_type_priority[3] = { GNUTLS_CRT_X509,
       GNUTLS_CRT_OPENPGP, 0};
    //initialize TLS session info
    gnutls_init (&socketInfo.session, GNUTLS_CLIENT);
       gnutls_set_default_priority (socketInfo.session);
    gnutls_certificate_type_set_priority (socketInfo.session,
       cert_type_priority); gnutls_certificate_allocate_credentials (&
       socketInfo.xcred);
    gnutls_credentials_set (socketInfo.session, GNUTLS_CRD_CERTIFICATE,
       socketInfo.xcred);

    //check if non-blocking. If so, make blocking until we are done with
        the handshake
    int socketflags = fcntl(*(int *)tjp->arg(0),F_GETFL);
    if ((socketflags & O_NONBLOCK) != 0) fcntl(*(int *)tjp->arg(0),
       F_SETFL, socketflags ^ O_NONBLOCK);
    //Connect + Handshake
    tjp->proceed();
    if (*tjp->result() < 0) {
          if ((socketflags & O_NONBLOCK) != 0)  fcntl(*(int *)tjp->arg
             (0),F_SETFL, socketflags);
      return;
    }
    gnutls_transport_set_ptr (socketInfo.session, (gnutls_transport_ptr)
        (*(int *)tjp->arg(0)));
    int result = gnutls_handshake (socketInfo.session);
    if ((socketflags & O_NONBLOCK) != 0){
      fcntl(*(int *)tjp->arg(0),F_SETFL, socketflags); //restore non-
         blocking state if it was like that
      gnutls_transport_set_lowat(socketInfo.session,0); //now make
         gnutls aware that we are dealing with non-blocking sockets
    }
    //Save Information in hash table
    socketInfo.isSecure = true; socketInfo.socketDescriptor = *(int *)
       tjp->arg(0);
    hardening_storeSocketInfo(*(int *)tjp->arg(0), socketInfo);
    *tjp->result() = result;
  }
```

Listing 6.4: Aspect for Adding HTTPS Functionality (Part 1)

```
        //replacing write() by gnutls_record_send() on a secured socket
  advice call("% write(...)") : around () {
    hardening_sockinfo_t socketInfo = hardening_getSocketInfo(*(int *)
        tjp->arg(0));
    if (socketInfo.isSecure)
       *(tjp->result()) = gnutls_record_send(socketInfo.session, *(char
          **) tjp->arg(1), *(int *)tjp->arg(2));
    else
       tjp->proceed();
  }

  //replacing read() by gnutls_record_recv() on a secured socket
  advice call("% read(...)") : around () {
    hardening_sockinfo_t socketInfo = hardening_getSocketInfo(*(int *)
        tjp->arg(0));
    if (socketInfo.isSecure)
       *(tjp->result()) = gnutls_record_recv(socketInfo.session, *(char
          **) tjp->arg(1), *(int *)tjp->arg(2));
    else
       tjp->proceed();
  }

  advice call("% close(...)") : around () {
    hardening_sockinfo_t socketInfo = hardening_getSocketInfo(*(int *)
        tjp->arg(0)); /* socket matched by sd*/
    if(socketInfo.isSecure ){
       gnutls_bye(socketInfo.session, GNUTLS_SHUT_RDWR);
    }
        tjp->proceed();
    if(socketInfo.isSecure ){
       gnutls_deinit(socketInfo.session);
          gnutls_certificate_free_credentials(socketInfo.xcred);
       hardening_removeSocketInfo(*(int *)tjp->arg(0));
       socketInfo.isSecure = false; socketInfo.socketDescriptor = 0;
    }
  }

};
```

Listing 6.5: Aspect for Adding HTTPS Functionality (Part 2)

```
--- C:/Documents and Settings/marc-andre/Desktop/apt-0.5.28.6-hardened/methods/http.cc
    Thu Apr 19 22:31:00 2007
+++ C:/Documents and Settings/marc-andre/Desktop/apt-0.5.28.6-hardened/methods/
    https2_orig_rework.cc    Thu Apr 19 22:39:01 2007
@@ -27,21 +26,23 @@
-bool CircleBuf::Read(int Fd)
+bool CircleBuf::Read(int Fd, gnutls_session_t session)
 {
        // Write the buffer segment
        int Res;
+       if (session == NULL) //added to deal with SSL
        Res = read(Fd,Buf + (InP%Size),LeftRead());
+       else
+        Res = gnutls_record_recv(session, Buf + (InP%Size),LeftRead());
@@ -98,11 +99,11 @@
-bool CircleBuf::Write(int Fd)
+bool CircleBuf::Write(int Fd, gnutls_session_t session)
 {
@@ -110,14 +111,16 @@
        // Write the buffer segment
        int Res;
+       if (session == NULL)
        Res = write(Fd,Buf + (OutP%Size),LeftWrite());
+       else //for SSL support
+        Res = gnutls_record_send(session,Buf + (OutP%Size),LeftWrite());
@@ -255,23 +259,45 @@
    // Connect to the remote server
-   if (Connect(Host,Port,"http",80,ServerFd,TimeOut,Owner) == false)
+   if (Connect(Host,Port,"http",443,ServerFd,TimeOut,Owner,session,xcred) == false)
       return false;
    return true;
 }
 bool ServerState::Close()
 {
+   //make sure that the socket is in blocking mode before closing it
+   int socketflags = fcntl(ServerFd,F_GETFL);
+   if ((socketflags & O_NONBLOCK) != 0){
+     socketflags ^= O_NONBLOCK;
+     fcntl(ServerFd,F_SETFL, socketflags);
+   }
+  if (session != NULL)
+   gnutls_bye(session, GNUTLS_SHUT_RDWR); //gnutls close sesion
    close(ServerFd); //close socket descriptor
+  if (session != NULL){
+   gnutls_deinit(session);
+   gnutls_certificate_free_credentials(xcred);
+   session = NULL;
+   xcred = NULL;
+  }
    ServerFd = -1;
    return true;
 }
```

Listing 6.6: Patch for Adding HTTPS Functionality from `http.cc` (part 1)

```
 // ServerState::RunHeaders - Get the headers before the data          /*{{{*/
@@ -685,26 +711,26 @@
    // Handle server IO
    if (Srv->ServerFd != -1 && FD_ISSET(Srv->ServerFd,&rfds))
    {
       errno = 0;
-      if (Srv->In.Read(Srv->ServerFd) == false)
+      if (Srv->In.Read(Srv->ServerFd, Srv->session) == false)
         return ServerDie(Srv);
    }
    if (Srv->ServerFd != -1 && FD_ISSET(Srv->ServerFd,&wfds))
    {
       errno = 0;
-      if (Srv->Out.Write(Srv->ServerFd) == false)
+      if (Srv->Out.Write(Srv->ServerFd, Srv->session) == false)
         return ServerDie(Srv);
    }
    // Send data to the file
    if (FileFD != -1 && FD_ISSET(FileFD,&wfds))
-      if (Srv->In.Write(FileFD) == false)
+      if (Srv->In.Write(FileFD,NULL) == false)
         return _error->Errno("write",_("Error writing to output file"));
    }
// Handle commands from APT
    if (FD_ISSET(STDIN_FILENO,&rfds))
@@ -727,12 +755,12 @@
       while (Srv->In.WriteSpace() == true)
       {
-        if (Srv->In.Write(File->Fd()) == false)
+        if (Srv->In.Write(File->Fd(),NULL) == false)
           return _error->Errno("write",_("Error writing to file"));
         if (Srv->In.IsLimit() == true)
           return true;
       }
@@ -752,12 +780,12 @@
    // Dump the buffer to the file
    if (Srv->State == ServerState::Data)
    {
       SetNonBlock(File->Fd(),false);
       while (Srv->In.WriteSpace() == true)
       {
-        if (Srv->In.Write(File->Fd()) == false)
+        if (Srv->In.Write(File->Fd(),NULL) == false)
           return _error->Errno("write",_("Error writing to the file"));
         // Done
         if (Srv->In.IsLimit() == true)
           return true;
@@ -940,10 +968,12 @@
 int HttpMethod::Loop()
 {
    signal(SIGTERM,SigTerm); signal(SIGINT,SigTerm);
+   gnutls_global_init();
@@ -1115,20 +1146,24 @@
+   gnutls_global_deinit();
    return 0;
 }
```

Listing 6.7: Patch for Adding HTTPS Functionality from `http.cc` (part 2)

```
--- C:/Documents and Settings/marc-andre/Desktop/apt-0.5.28.6-hardened/methods/connect.cc
           Wed Mar 14 15:59:02 2007
+++ C:/Documents and Settings/marc-andre/Desktop/apt-0.5.28.6-hardened/methods/connectssl2
    .cc   Thu Apr 19 10:00:10 2007
@@ -52,3 +62,4 @@
 static bool DoConnect(struct addrinfo *Addr,string Host,
-                      unsigned long TimeOut,int &Fd,pkgAcqMethod *Owner)
+                      unsigned long TimeOut,int &Fd,pkgAcqMethod *Owner, gnutls_session_t
    &session,
+                      gnutls_certificate_credentials &xcred)
 {
@@ -83,6 +94,56 @@
    SetNonBlock(Fd,true);
+       static const int cert_type_priority[3] = { GNUTLS_CRT_X509, GNUTLS_CRT_OPENPGP,
    0};
+       //initialize TLS session info
+   gnutls_init (&session, GNUTLS_CLIENT);
+       gnutls_set_default_priority (session);
+       gnutls_certificate_type_set_priority (session, cert_type_priority);
+       gnutls_certificate_allocate_credentials (&xcred);
+       gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);
+       int socketflags = fcntl(Fd,F_GETFL);
+       if ((socketflags & O_NONBLOCK) != 0){
+               fcntl(Fd,F_SETFL, socketflags ^ O_NONBLOCK);
+       }
    if (connect(Fd,Addr->ai_addr,Addr->ai_addrlen) < 0 &&
-       errno != EINPROGRESS)
-      return _error->Errno("connect",_("Cannot initiate the connection "
-                           "to %s:%s (%s)."),Host.c_str(),Service,Name);
+       errno != EINPROGRESS){
+               if ((socketflags & O_NONBLOCK) != 0){
+                       fcntl(Fd,F_SETFL, socketflags);
+               }
+          return _error->Errno("connect",_("Cannot initiate the connection to %s:%s (%s).
    "),Host.c_str(),Service,Name);
+   }
+       gnutls_transport_set_ptr (session, (gnutls_transport_ptr) Fd);
+       int result = gnutls_handshake (session);
+       if ((socketflags & O_NONBLOCK) != 0){
+               fcntl(Fd,F_SETFL, socketflags);
+               gnutls_transport_set_lowat(session,0); //now make gnutls aware that we are
     dealing with non-blocking sockets
+       }
+       if (result < 0) return false;
@@ -114,3 +175,4 @@
 bool Connect(string Host,int Port,const char *Service,int DefPort,int &Fd,
-            unsigned long TimeOut,pkgAcqMethod *Owner)
+            unsigned long TimeOut,pkgAcqMethod *Owner, gnutls_session_t &session,
+                     gnutls_certificate_credentials &xcred)
 {
@@ -185,3 +247,3 @@
    {
-      if (DoConnect(CurHost,Host,TimeOut,Fd,Owner) == true)
+      if (DoConnect(CurHost,Host,TimeOut,Fd,Owner, session, xcred) == true)
       {
```

Listing 6.8: Patch for Adding HTTPS Functionality from `connect.cc`

Chapter 7

Conclusion

This thesis summarizes two years of intense and stimulating learning and reflection on the ideal manner to achieve this nearly Utopian dream of automatic security hardening. Many hours of programming were spent trying to find the ideal, minimally-intrusive, way to harden even simple programs, and to reflect upon many scenarios that could make things go wrong. Those hours of challenging programming provoked an increased respect for security professionals and a confirmation that this research is a necessary evolution in the field. The many hours went into building good algorithms that would facilitate the practical implementation of our contributions proved to be a challenge in learning, understanding, expression and conciseness.

Throughout this thesis, we saw that manual security hardening practices performed by experts suffered from pragmatic limitations that justify the creation of a new approach which will allow non-expert maintainers improve the security of existing software by modifying the source. We learnt about the nature of software security and hardening practices, and saw that AOP could be integrated in a synergic manner with patterns, resulting in

security hardening patterns. We offered many new contributions to the field of software security: a classification of hardening practices, an evaluation of security design patterns and a study of Linux kernel hardening practices.

Building on this first layer of knowledge, we established an approach that would enable laypersons to perform security hardening by combining security hardening plans, security hardening patterns, *SHL*, and security hardening aspect refinement. This approach showed a separation of concerns, where security experts would create patterns and developers would specify hardening plans and refine them into security solutions ready to be automatically injected in their source code. We also contributed to the theoretical richness of AOP languages by designing four primitives that enabled parameter passing and relative pointcut determination.

Once this other layer was established, we were now able to create an AOP-based language, *SHL*, designed to allow the simple and effective specification of hardening patterns and plans in a language-independent manner. We then used this language to write a hardening plan and a pattern in order to add HTTPS support to APT, the Debian package manager and downloader. This case study confirmed our intuition that our approach would lower the level of digging necessary to harden an application, would lower the time required in order to make the necessary modifications, would enable reuse and would require very little expertise to accomplish.

We thus demonstrated by both theory and practice that it is possible to leverage AOP-based patterns in order to systematically apply security hardening solutions without the need to know the security domain nor the specific APIs used. This approach solves many problems observed in the world of patterns and security design patterns and paves the way

for automatic security hardening in the future.

Future work is likely to center around the development of automated refining as a translation between *SHL* the target AOP language and a target build system. Research will be required in order to enrich *SHL* with the ability to have behavior code be determined according to plan parameters. This will mean new primitives such as `IF`, `SWITCH`, `GETPLANPARAMETER`, etc. It is also possible that more patterns and APIs will be developed, which is likely to enrich the world of AOP and the vocabulary in *SHL*. My colleagues in the Computer Security Laboratory will also need to consider how to correctly and automatically map between different types of error codes, so that error handling happens smoothly in situation that would otherwise see incompatibility. As a software security specialist, it is my hope to see this approach becoming an available system in the future, and that this system will be used all over the world to drastically transform the state of software security for the better.

Bibliography

[1] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.

[2] Dima Al-Hadidi, Nadia Belblidia, and Mourad Debbabi. Security crosscutting concerns and AspectJ. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006).* McGraw-Hill/ACM Press, 2006.

[3] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language.* Oxford University Press, 1977.

[4] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Trans. Program. Lang. Syst.*, 11(1):115–146, 1989.

[5] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.

[6] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.

[7] Matt Bishop. How Attackers Break Programs, and How to Write More Secure Programs. http://nob.cs.ucdavis.edu/~bishop/secprog/sans2002/index.html (accessed 2007/04/19).

[8] Matt Bishop. Writing safe secure programs, 1997. http://nob.cs.ucdavis.edu/bishop/secprog/ns1997.pdf (accessed 2007/04/19).

[9] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2002.

[10] BitKeeper. Bitkeeper, 2005. http://linux.bkbits.net/ (accessed 2007/04/18).

[11] Bob Blakley, Craig Heath, and members of The Open Group Security Forum. Security design patterns. Technical Report G031, Open Group, 2004.

[12] Ron Bodkin. Enterprise security aspects. In *Proceedings of the AOSD 04 Workshop on AOSD Technology for Application-level Security (AOSD'04:AOSDSEC)*, 2004.

[13] Jonas Bonér. Semantics for a synchronized block join point, 2005. http://jonasboner.com/2005/07/18/semantics-for-a-synchronized-block-joint-point/ (accessed 2007/04/19).

[14] Alexandre M. Braga, Cecília M. F. Rubira, and Ricardo Dahab. Tropyc: A pattern language for cryprographic software. Technical Report IC-99-03, Institute of Computing, UNICAMP, January 1999.

[15] David Callahan, Alan Carle, Mary Wolcott Hall, and Ken Kennedy. Constructing the procedure call multigraph. *IEEE Trans. Softw. Eng.*, 16(4):483–487, 1990.

[16] Andrei Soeanu Caval. Automatic verification of behavioral specifications in software-intensive systems. Master's thesis, Concordia University, 2007.

[17] Shuo Chen, Zbigniew Kalbarczyk, Jun Xu, and Ravishankar K. Iyer. A data-driven finite state machine model for analyzing security vulnerabilities. In *2003 International Conference on Dependable Systems and Networks (DSN'03)*, page 605. IEEE, 2003.

[18] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[19] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[20] Bart DeWin. *Engineering Application Level Security through Aspect Oriented Software Development.* PhD thesis, Katholieke Universiteit Leuven, 2004.

[21] F. Lee Brown, Jr., James DiVietri, Graziella Diaz de Villegas, and Eduardo B. Fernandez. The authenticator pattern. In *Proceedings of the 6th Annual Conference on the Pattern Languages of Programs (PLoP99)*, 1999.

[22] Eduardo B. Fernandez and Reghu Warrier. Remote authenticator/authorizer. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP 2003)*, 2003.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[24] Mark G. Graff and Ken van Wyk. *Secure Coding: Principles and Practices*. O'Reilly & Associates, 2003.

[25] Open Source Technology Group. Sourceforge.net: Software map. `http://sourceforge.net/softwaremap/trove_list.php` (accessed 2006/02/01).

[26] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[27] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, 1997.

[28] Steve Hamm. Linux inc. *BusinessWeek Online*, 01 2001. `http://www.businessweek.com/magazine/content/05_05/b3918001_mz001.htm` (accessed 2007/04/19).

[29] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005), March*, 2005.

[30] Michael Howard and David E. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2002.

[31] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.

[32] Minhuan Huang, Chunlei Wang, and Lufeng Zhang. Toward a reusable and generic security aspect library. In *Proceedings of the AOSD 04 Workshop on AOSD Technology for Application-level Security (AOSD'04:AOSDSEC)*, 2004.

[33] IDC. Server market momentum continues as consolidation and virtualization drive it infrastructure investment, according to idc, 05 2007. `http://www.idc.com/getdoc.jsp?containerId=prUS20699807` (accessed 2007/06/14).

[34] Gregor Kiczales. The fun has just begun, keynote talk at AOSD 2003, 2003. `http://www.cs.ubc.ca/~gregor/papers/kiczales-aosd-2003.ppt` (accessed 2007/04/19).

[35] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[36] Darrell M. Kienzle and Matthew C. Elder. Final technical report: Security patterns for web application development. Technical Report DARPA Contract # F30602-01-C-0164, 2002. `http://www.modsecurity.org/archive/securitypatterns/dmdj_final_report.pdf` (accessed 2007/04/19).

[37] Darrell M. Kienzle, Matthew C. Elder, David Tyree, and James Edwards-Hewitt. Security patterns repository, 2002. `http://www.modsecurity.org/archive/securitypatterns/dmdj_repository.pdf` (accessed 2007/04/19).

[38] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'73)*, pages 194–206, New York, NY, USA, 1973. ACM Press.

[39] Ulf Lamping, Richard Sharpe, and Ed Warnicke. Wireshark User's Guide. `http://www.wireshark.org/download/docs/user-guide-us.pdf` (accessed 2007/06/15).

[40] Marc-André Laverdière, Azzam Mourad, Aiman Hanna, and Mourad Debbabi. Security design patterns: Survey and evaluation. In *IEEE Canadian Conference on Electrical and Computer Engineering*. IEEE Press, 2006.

[41] M-A. Laverdière, A. Mourad, A. Soeanu, and Mourad Debbabi. Control flow based pointcuts for security hardening concerns. In *To appear in the Proceedings of the 2007 International Conference on Privacy, Security and Trust (IFIP-PST 2007)*. Springer, 2007.

[42] Marc-André Laverdière, Azzam Mourad, Andrei Soeanu, and Mourad Debbabi. Points de coupure pour les préoccupations de renforcement de sécurité utilisant le flot de contrôle. In *Actes de la Troisièmme Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2007)*, pages 45–60, 2007.

[43] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.

[44] David Mackenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files*. Free Software Foundation, 2002. `http://www.gnu.org/software/diffutils/manual/ps/diff.ps.gz` (accessed 2007/04/19).

[45] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03)*, pages 105–121, 2003.

[46] Nikos Mavroyanopoulos and Simon Josefsson. *GNU TLS: Transport Layer Security Library for the GNU system for version 1.7.13, 31 May 2007*. Free Software Foundation, 2007. `http://www.gnu.org/software/gnutls/manual/gnutls.pdf` (accessed 2007/06/14).

[47] Azzam Mourad, Marc-André Laverdière, and M. Debbabi. Security hardening of open source software. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006)*. McGraw-Hill/ACM Press, 2006.

[48] Azzam Mourad, Marc-André Laverdière, and M. Debbabi. A high-level aspect-oriented language for software security hardening. In *Proceedings of the International Conference of Security and Cryptography (ICETE-SECRYPT 2007)*, 2007.

[49] Azzam Mourad, Marc-André Laverdière, and Mourad Debbabi. New primitives to AOP weaving capabilities for security hardening concerns. In *Proceedings of the 9th International Conference on Enterprise Information Systems*. ICEIS, 2007.

[50] Azzam Mourad, Marc-André Laverdière, and Mourad Debbabi. Towards an aspect oriented approach for the security hardening of code. *To appear in Elsevier Computers & Security*, 2007.

[51] Azzam Mourad, Marc-André Laverdière, and Mourad Debbabi. Towards an aspect oriented approach for the security hardening of code. In *Proceedings of the 3rd IEEE International Symposium on Security in Networks and Distributed Systems (IEEE AINA-SSNDS 2007)*. IEEE Press, 2007.

[52] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '99)*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[53] NIST. National vulnerability database, 2007. `http://nvd.nist.gov/` (accessed 2007/04/18).

[54] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10):43–50, 2001.

[55] Terence Parr. Antlr. `http://www.antlr.org` (accessed 2007/04/19).

[56] Torsten Priebe, Eduardo B. Fernandez, Jens I. Mehlau, and Günther Pernul. A pattern system for access control. In *Proceedings 18th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, 2004.

[57] Sasha Romanosky. Security design patterns part 1, 2001. `http://www.romanosky.net/` (accessed in 2005).

[58] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):216– 226, 1979.

[59] Markus Schumacher. *Security Engineering with Patterns*. Springer, 2003.

[60] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley, 2006.

[61] Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 13–22. IEEE, 2005.

[62] Robert C. Seacord. *Secure Coding in C and C++*. SEI Series. Addison-Wesley, 2005.

[63] Viren Shah. An aspect-oriented security assurance solution. Technical Report AFRL-IF-RS-TR-2003-254, Cigital Labs, 2003.

[64] Viren Shah and Frank Hill. Using aspect-oriented programming for addressing security concerns. In *Procceedings of the Thirteenth International Symposium on Software Reliability Engineering (ISSRE)*, pages 115–119, 2002.

[65] Robert W. Shirey. Internet Security Glossary. RFC 2828 (Informational), May 2000.

[66] Pawel Slowikowski and Krzysztof Zielinski. Comparison study of aspect-oriented and container managed security. In *Proceedings of the ECCOP workshop on Analysis of Aspect-Oriented Software*, 2003.

[67] Sopoforic. Image:partitionlattice.svg. Wikimedia Commons. http://commons.wikimedia.org/wiki/Image:PartitionLattice.svg (accessed 2007/04/17).

[68] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific (CRPIT '02)*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[69] Gary Stoneburner, Clark Hayden, and Alexis Feringa. Engineering principles for information technology security (a baseline for achieving security), revision A. Technical Report Special Publication 800-27 Rev A, NIST, 2004.

[70] Dimitri van Heesch. *Doxygen Manual for version 1.5.2*. Doxygen, 2007. ftp://ftp.stack.nl/pub/users/dimitri/doxygen_manual-1.5.2.pdf.zip (accessed 2007/04/17).

[71] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive programming in JAsCo. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM Press.

[72] John Viega and Matt Messier. *Secure Programming Cookbook For C and C++*. O'Reilly Media, Inc., 2003.

[73] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010.* 2003. `http://www.dwheeler.com/secure-programs/` (accessed 2007/04/19).

[74] Ratsameetip Wita and Yunyong Teng-Amnuay. Vulnerability profile for linux. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications*, pages 953–958. IEEE, 2005.

[75] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *Proceedings of the 4th Annual Conference on the Pattern Languages of Programs (PLoP97)*, 1997.