SECURITY ON DEMAND

BY

MUNAWAR HAFIZ

DISSertation

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Research Associate Professor Ralph Johnson, Chair and Director of Research
Professor Carl Gunter
Associate Professor Sam Kamin
Assistant Professor Darko Marinov
Professor David Garlan, Carnegie-Mellon University
Abstract

Security experts generally believe that, “security cannot be added on, it must be designed from the beginning.”

This dissertation shows that retrofitting security does not need to be a massive reengineering effort, nor does it need to be ad hoc. Security solutions can be added through systematic, general purpose security-oriented program transformations. We describe a catalog of security-oriented program transformations; so far the catalog contains thirty seven transformations. These security-oriented program transformations improve the traditional approaches of security engineering and keep software secure in the face of new security threats. Security-oriented program transformations are not silver bullets; using them requires skill and knowledge of the program being transformed. They are instead power tools that make it easier to add security to existing systems replacing the point solution of a typical patch with a more systematic removal of a vulnerability. When appropriate tools are built and the program transformations are easy enough to apply, then they will allow a software developer to add ‘security on demand’.
To My Family.
I thank my wife Farhana Ashraf for being there all the time. She has made available her support in a number of ways.

I have immense gratitude for my father, Mohammad Abdul Hafiz, and my mother, Nargis Hafiz, for their dedication and confidence in me. My sister, Fariha Sultana, always had for me words of encouragement.

I am indebted to many of my colleagues at Software Architecture Group and Security Group at University of Illinois. Thank you guys for being wonderful friends.

Finally, I thank my committee members Dr. Ralph Johnson (UIUC), Dr. Carl Gunter (UIUC), Dr. Sam Kamin (UIUC), Dr. Darko Marinov (UIUC), and Dr. David Garlan (CMU) for patiently persevering with me.
# Table of Contents

List of Tables .......................................................... ix

List of Figures .......................................................... x

Chapter 1  Introduction .................................................. 1
  1.1 Security-oriented Program Transformations ........................ 3
  1.1.1 Thesis Statement .............................................. 3
  1.1.2 Elaboration of Claims ........................................ 4
  1.1.3 Summary of Contributions .................................... 7
  1.2 Plan of Dissertation ............................................. 8

Chapter 2  Software Evolution Case Studies ............................. 9
  2.1 Evolution of Mail Transfer Agents ................................. 9
  2.1.1 From sendmail to MeTa1 ..................................... 10
  2.1.2 Changes in Mail Transfer Agent Architecture .................. 11
  2.2 Evolution of Microsoft Word ..................................... 13
  2.3 Lessons Learned .................................................. 15

Chapter 3  Security Engineering Practices: A History of Failed Attempts to Delay Software Aging ............................................ 17
  3.1 Secure Software Development Approaches ........................ 18
  3.1.1 Defining Standards for Secure Software Development ............ 19
  3.1.2 Integrating Security During Software Development Lifecycle .... 20
  3.1.3 Being Agile .................................................... 22
  3.2 Up Front Security Design ......................................... 22
  3.3 Secure Software Maintenance Approaches .......................... 23
  3.4 Aging and Secure Software Development Approaches ............... 24
  3.5 Aging and Secure Software Maintenance Approaches ............... 26
  3.6 Summary .......................................................... 27

Chapter 4  Security-oriented Program Transformations .................. 28
  4.1 Security-oriented Program Transformations ....................... 28
  4.2 Related Work ..................................................... 33
  4.3 Security Engineering Using Program Transformations .............. 35
  4.4 Advantages of New Security Engineering ........................... 37
  4.5 Applying Program Transformations ................................ 37
  4.6 Issues of Applying Program Transformations ...................... 39
  4.7 Summary .......................................................... 40
# Limiting Authentication Attempts to Prevent Brute Force Attacks

8.4.1 *Add Account Lockout* Transformation ........................................... 132

# Introducing Single Sign On Components

8.5.1 *Credential Tokenizer* Transformation ........................................... 133
8.5.2 *Secure Session Object* Transformation .......................................... 136
8.5.3 *Add Password Synchronizer* Transformation .................................. 137
8.5.4 *Single Sign On Delegate* Transformation ....................................... 139

# Organizing the Transformations

8.6.1 Impact on Codebase ................................................................. 142
8.6.2 Type of Transformation ............................................................ 142
8.7 Composing the Transformations ..................................................... 143
8.8 Summary ......................................................................................... 144

## Chapter 9 Security-oriented Program Transformations for Access Control

9.1 Broken Access Control Vulnerability .................................................. 145
9.3 Adding Authorization Components .................................................. 147
9.3.1 *Add Authorization Enforcer* Transformation ............................... 147
9.4 Regulating How Privilege is Passed during Process or Object Creation
9.4.1 *Controlled Process Creation* Transformation ............................... 150
9.4.2 *Guarded Object* Transformation ................................................ 153
9.4.3 *Secure Resource Pooling* Transformation .................................... 156
9.5 Protecting Resources ................................................................. 159
9.5.1 *Encryption/ Decryption* Transformation ....................................... 159
9.5.2 *Message Digest Creation* Transformation .................................... 160
9.5.3 *Signature Generation* Transformation .......................................... 162
9.6 Organizing the Transformations ..................................................... 164
9.6.1 Impact on Codebase ................................................................. 165
9.6.2 Type of Transformation ............................................................ 165
9.7 Composing the Transformations ..................................................... 166
9.8 Summary ......................................................................................... 166

## Chapter 10 Security-oriented Program Transformations for Proper Error Handling

10.1 Improper Error Handling Vulnerability .......................................... 167
10.3 Adding Components for Storing State Information ......................... 169
10.3.1 *Add Audit Interceptor* Transformation ....................................... 169
10.3.2 *Secure Logger* Transformation ................................................ 171
10.4 Restarting Gracefully from Error .................................................... 174
10.4.1 *Checkpointed System* Transformation ....................................... 174
10.5 Handling Error Messages ............................................................ 175
10.5.1 *Error Message Suppressor* Transformation ............................... 175
10.5.2 *Exception Shielding* Transformation .......................................... 178
10.6 Organizing the Transformations ..................................................... 181
10.7 Composing the Transformations ..................................................... 182
10.8 Summary ......................................................................................... 182
Chapter 11 Security-oriented Program Transformations for Ensuring Availability . . . . . . . 183
  11.1 Denial of Service Vulnerability ........................................... 183
  11.2 Solution Strategies ............................................................... 184
  11.3 Adding Components for Limiting Incoming Requests ......................... 184
      11.3.1 Message Caching Transformation .................................... 184
  11.4 Using Resources Parsimoniously ........................................... 186
      11.4.1 Resource Management Transformation ................................. 186
  11.5 Throwing in Redundancy ...................................................... 190
      11.5.1 Replicated Component Transformation ............................... 190
      11.5.2 Standby Transformation ............................................... 193
      11.5.3 Tandem Component Transformation .................................. 196
  11.6 Organizing the Transformations ............................................ 199
  11.7 Composing the Transformations ............................................ 200
  11.8 Summary ........................................................................ 200

Chapter 12 Defense In Depth with Program Transformations .............................. 201
  12.1 Preventing Buffer Overflow Vulnerability in C Programs .................... 201
      12.1.1 Replacing Unsafe Libraries ........................................... 202
      12.1.2 Enforcing Correct Integer Operation ................................ 206
      12.1.3 Replacing Pointers with Safe Data Structure ........................ 207
      12.1.4 Input Rectification and Other Techniques .......................... 208
  12.2 Preventing SQL Injection Vulnerability in PHP Programs .................... 209
      12.2.1 Replacing Unsafe Libraries ........................................... 210
      12.2.2 Applying Filters for Input Rectification .............................. 212
  12.3 Conclusion ........................................................................ 213

Chapter 13 Usefulness of the Program Transformation Catalog ........................... 214
  13.1 Coverage of Program Transformations ...................................... 214
  13.2 Adoption of Program Transformations ...................................... 216
  13.3 Conclusion ........................................................................ 221

Chapter 14 Conclusion ........................................................................ 222
  14.1 Open Research Questions ....................................................... 222
  14.2 Future Work ....................................................................... 223
  14.3 Onward! ............................................................................ 225

Appendix A Catalog of Security-oriented Program Transformations ................. 226

References .............................................................................. 233
List of Tables

5.1 Classification of security patterns using the Enterprise Architectural Space Organizing Table 49
6.1 Fundamental Security-oriented Program Transformations 77
7.1 Sample policies for various transformations 92
7.2 Some unsafe functions and their safe alternatives 100
7.3 Patterns for strcat and strcpy functions 101
7.4 Security-oriented Program Transformations for Unvalidated Input 123
8.1 Security-oriented Program Transformations for Broken Authentication 142
9.1 Security-oriented Program Transformations for Broken Access Control 165
10.1 Security-oriented Program Transformations for Improper Error Handling 181
11.1 Security-oriented Program Transformations for Denial of Service 199
13.1 Vulnerability Trends in 5 Random Weeks 215
## List of Figures

2.1 Main Components of an MTA .................................................. 11

4.1 Schematic Diagram of a Security-oriented Program Transformation .............. 29

5.1 Hierarchical Classification Scheme ........................................... 50
5.2 Finding security patterns that are related to software .............................. 52
5.3 Finding candidate security patterns ............................................ 53
5.4 List of security-oriented program transformations .................................. 55

6.1 Class diagram showing important classes and methods of jftpd ...................... 73
6.2 Class diagram showing the partitions ........................................... 74
6.3 Classes added by a *Policy Enforcement Point* transformation .................... 76

7.1 Comfort zone of inputs and input rectification .................................... 79
7.2 Applying *Add Perimeter Filter* transformation .................................... 84
7.3 Input rectification process ..................................................... 85
7.4 Sequence diagram showing how a message interceptor gateway works ............. 87
7.5 Class diagram describing policies to apply on an unsafe string .................... 90

8.1 Classes added by an *Add Authentication Enforcer* transformation ............... 130
8.2 Classes added by a *Credential Tokenizer* transformation ........................ 135
8.3 New synchronizer service created by an *Add Password Synchronizer* transformation .... 139
8.4 Important classes of the delegator component ..................................... 141

9.1 Steps of authorization introduced by the *Add Authorization Enforcer* transformation ...... 149
9.2 A guard object encapsulating an unprotected resource as created by an *Guarded Object* transformation .................................................. 155

10.1 Processing audit data with an audit interceptor component ...................... 170
10.2 Encryption and Signature added as a *Decorated Filter* transformation .......... 173
10.3 Adding suppression policies to an error message .................................. 177
10.4 Inheritance hierarchy of the newly defined exceptions ............................ 180

11.1 Classes introduced by a *Replicated Component* transformation ................. 192
11.2 Classes introduced by a *Standby* transformation .................................. 195
11.3 Classes introduced by a *Tandem Component* transformation ....................... 198

13.1 Feasibility of Building Program Transformation Tools ........................... 217
13.2 Security-oriented Program Transformations That Involve Connector Transformations .... 219
Chapter 1

Introduction

A software application that defies change is destined to fail. Changes come in various forms: end users ask for new features to be added, new development in hardware and operating systems call for porting an application to a new platform, new demands for performance warrant a change in the underlying algorithm, etc. In this work, we are interested in changes that secure an application in keeping with the change in security requirements.

The change in security requirements manifests itself in two ways. First, application vendors can decide that they want to add new security protection to a system as a new feature. Second, attackers may come up with new security exploits forcing application developers to add preventive measures. Hence, vendors can take the decision to upgrade security as early as when they are analyzing the requirements for a new application, to as late as when they want to patch an already released application. Adapting an application in both cases is mostly an ad hoc, manual activity.

This ad hoc nature of security engineering gives an attacker an advantage. Keeping systems secure is a game with a moving target. There are no bounds on either the creativity of attackers in finding new vulnerabilities, or the creativity of secure system developers in writing patches to remove them. Since the same vulnerability affects many types of software, attackers can use various tools and be systematic in their approach. In contrast, the preventive measures for these vulnerabilities are added manually and in an ad hoc manner. For example, we have surveyed Bugtraq [34] list on a random week (the first week of September 2008), and counted 28 instances of buffer overflow vulnerabilities affecting software from 22 different vendors. At least 17\textsuperscript{1} of these incidents occurred due to the use of unsafe functions; the common fix is to validate the inputs or replace the functions. Despite the common problem and solutions, there are no general purpose tools that developers can use to introduce the solution. The ad hoc fixes, therefore, vary

\textsuperscript{1}For the remaining 11 cases, the source code or the exploit details were unavailable. We could not identify the root cause of these vulnerabilities.
from case to case; their effectiveness depends on the experience and the vision of the developers.

Ad hoc security engineering has other problems\(^2\). Often developers work under time pressure and can only concentrate on solving the vulnerability that has been reported. As a result, they produce local fixes; the same problem may remain in the application. For example, Buqtraq [34] lists 39 instances of data injection vulnerabilities in Microsoft Word 2000. Each time, Microsoft developers created a patch to remove the specific vulnerability, only to let the same vulnerability resurface later in another part of the program.

Because there is no systematic approach to adapt an application to meet changing security requirements, the retrofitting process has gotten a bad reputation. Security experts, from their painful experience in trying to retrofit security, believe that “security cannot be added on, it must be designed from the beginning” [7]. But consider redesigning Microsoft Word from scratch everytime a new security vulnerability is found; for any meaningful, large software this redesign approach does not scale. Retrofitting security is the only feasible option, but it must be done systematically.

Having systematic tools and techniques to introduce security protection has its advantages. These techniques can be applied to retrofit security to an existing system; but more importantly, they will be an integral part of designing and implementing secure systems. Developers can use these tools and techniques during software development to try alternative security solutions and find the most suitable one.

Our work explores how security can be ‘added on’ systematically. We show that retrofitting security to an existing system does not have to be a massive reengineering effort, nor does it need to be ad hoc. Instead, it is possible to improve the security of a system by applying systematic, general purpose security-oriented program transformations.

There are many kinds of program transformations. Compilers transfer programs in source form to equivalent programs for a particular machine language. Refactorings are source to source transformations that change the structure of programs but not their behavior. Security-oriented program transformations include both source to source and binary to binary transformations. They improve the security of systems, which means that they do not preserve all types of behavior. They preserve expected behavior, but should change a system’s response to security attacks.

A software system is often thought of as a collection of components. In that case, “adding on” to a software system means adding a new component. However, a software system can also be thought of

\(^2\)We will describe the problems of ad hoc security engineering in more detail in chapter 3.
as a sequence of program transformations. For example, consider a sequence of versions in a version control system, from the earliest version to the latest. The difference between any two adjacent versions is a program transformation. Some version control systems, such as CVS, store the differences between versions instead of storing the versions directly. These systems produce a version by applying a series of program transformations. These program transformations are not general purpose, but they show that a program can be thought of as a sequence of transformations.

Security-oriented program transformations, on the other hand, are general purpose. They are platform specific and language specific; but a program transformation for a specific platform and a specific language can be applied to any program written in that language and running in that platform. We have prepared a catalog of security-oriented program transformations; so far the catalog contains 37 program transformations. These program transformations provide solutions for the most important security problems faced by today’s developers. This dissertation narrates the mechanism of program transformations and how the transformations can be automated. The availability of automated tools for program transformations will allow an application developer to add “security on demand”.

1.1 Security-oriented Program Transformations

A program transformation is a function that maps an input program to an output program. A security-oriented program transformation is a function that maps an input program to a security-augmented program, i.e., it adds some sort of security protection to an existing program.

1.1.1 Thesis Statement

My dissertation demonstrates that,

It is possible to think of applying security in terms of program transformations. The exact set of program transformations is flexible; this dissertation will show a set that is sufficient enough to solve the most important security problems. Furthermore, these program transformations can be composed to support defense in depth. By ensuring that security-oriented program transformations concentrate on making structural changes only, the transformations can be

3 From this point, we use the terms transformations, program transformations and security-oriented program transformations interchangeably.
automated by general purpose tools that apply the changes. Using program transformations allows a software developer or a software maintainer to add “security on demand”.

We have analyzed the solutions for well-known security problems and described how these solutions can be automated. Our work also describes how to apply the security-oriented program transformations, covering issues such as composition of security solutions to have defense in depth.

1.1.2 Elaboration of Claims

The thesis statement is composed of four separate claims.

First, our research shows that it is possible to think of applying security in terms of program transformations. The traditional belief about ‘applying security’ revolves around the idea that security is architectural; it is a property of the entire system and hence cannot be added by adding a module [62]. We agree to this belief, but we have also observed that we can make structural changes to a program, sometimes tantamount to making an architectural change, and add security protection. Our claim is that it is possible to improve the security of a system by applying program transformations.

What type of protection are we introducing? This thesis concentrates on security problems in the context of a software application, and how their solutions can be introduced as program transformations. An example problem is a buffer overflow attack on a C application. There are many types of mistakes that contribute to a buffer overflow vulnerability: one major cause is that C applications use unsafe library functions for handling buffers. A possible solution is to replace all unsafe functions in a C application with safe functions; a program transformation may describe the mechanism. Another example problem is systematically adding authentication. When a web application developer wants to add authentication support, he or she builds an authentication component manually, possibly using some library or authentication framework. A program transformation may describe how an authentication component can be composed using a library or an authentication framework; a general purpose tool can be built that uses a specific framework and composes an authentication component for all programs written in a specific programming language. However, many other security problems are not directly problems of software applications, e.g., a network-based denial of service attack, or a phishing attack. Our claim is not about these security problems.

How do we apply security? We have observed that the solutions to application security issues follow common change patterns that can be generalized. Most of the changes are about replacing an old, unsafe
artifact with a new, safe artifact, e.g., replacing a method call to protect from buffer overflow. In some cases, a program transformation composes classes to build a component, and delegates a task to the newly composed, secure component. Sometimes, a program transformation refactors a program to apply a design pattern [74]. Another technique is to refactor a program to distribute artifacts, e.g., partition a single process into multiple processes etc. Our claim is that these methods change program structure, a change that also introduces security protection.

Second, The exact set of program transformations is flexible; this dissertation will show a set that is sufficient enough to solve the most important security problems. Furthermore, these program transformations can be composed to support defense in depth. We have been maintaining a catalog of general purpose security-oriented program transformations that can be applied to solve the most important security problems faced by today's developers; so far the catalog contains 37 security-oriented program transformations. To derive the catalog, we have analyzed software security from the problem domain and solution domain. As part of the problem domain, we have studied various vulnerability trend reports and analyzed the threat model of various software applications to identify the most important security problems for current application developers. Once we have identified the relevant security problems, we have identified the security solutions. To find the solutions, we have used our catalog of security patterns [92], that describes security solutions that experts use to solve a problem in a context. But a security pattern only provides a high level description of a security solution. It does not describe how to implement the solution, let alone how to automate it as a program transformation. For each program transformation, we have to map the high level security solution to the implementation mechanism of a security-oriented program transformation.

We have organized the catalog based on the vulnerability fixed by the program transformations. We have identified 5 distinct types of vulnerabilities that are the most relevant for today’s software developers. Our program transformations solve these vulnerabilities: 1) Unvalidated Input, 2) Broken Access Control, 3) Broken Authentication, 4) Improper Error Handling, and 5) Denial of Service. Our program transformations solve the most important security problems in each of the vulnerability classes.

Applying one program transformation does not secure an application; one must apply several solutions in different layers [210]. The program transformations in our catalog are composable. Some program transformations are orthogonal; they can be composed in any order. Other program transformations are useful only if applied in a specific way. The description of each program transformation includes preconditions
that are helpful in determining the order of composing program transformations.

**Third.** The transformations can be automated by general purpose tools that apply the changes. General purpose security-oriented program transformation tools can be built since the transformations make *structural changes* only. By structural change, we mean two things. First, the program transformations modify the structure of code: replace method calls, compose classes, refactor to patterns, distribute artifacts, etc. But more importantly, structural change means that a program transformation can make these changes only by looking at the structure of a program, without deep understanding of the program behavior. This makes security-oriented program transformations similar to refactorings [71]. Security-oriented program transformation tools can be built similar to refactoring tools; in fact, many program transformation tools can be built entirely from existing refactorings.

An important observation that contributes to the automation is the “separation of policy from mechanism”. “Separation of policy from mechanism” is an often-used concept in computer science, and in each context the people using it have their own interpretations. We use the concept to differentiate the steps of applying a program transformation that are part of an automated tool from the steps that are done manually. When a developer wants to apply a security-oriented program transformation, he or she has to follow three steps – 1) identify the program points where a transformation will be applied, 2) determine which transformation to apply, and 3) use a tool to automatically transform the program. The first two tasks require human intervention, i.e., a developer, perhaps using some sort of vulnerability detection tool, has to identify where to apply a transformation and which transformation to apply. This specification is the *policy*. The tools implement the *mechanism*; they instrument the changes automatically. The separation ensures that a tool does not need to have a detailed understanding of application logic. It only concentrates on making structural changes to a program.

Some program transformations are fully automated, but most of them require some sort of specification. The amount of specification varies; program transformations that make large changes to the code typically require a lot of specification. These program transformation are interactive: developers implement some steps manually and use tools for others. Section 13.2 discusses program transformations and the feasibility of building tools for program transformations.

We describe the steps of a program transformation. We have also built some program transformation tools applicable to C and Java programs. These tools are research prototypes, not commercial-quality prod-
ucts. Nevertheless, we have applied the tools to remove real vulnerabilities from real applications.

**Fourth**, application developers can use security-oriented program transformations to add “security on demand”. ‘Security on demand’ means when an application developer needs more security for an application, he or she should be able to get it. This will be possible with the development of commercial-quality security-oriented program transformation tools.

If these tools are available, program transformations will improve the traditional approaches of security engineering. Developers will be able to use program transformation tools during application development in the same way they use refactoring tools. Furthermore, the tools will make it easier for them to test new security alternative solutions and choose among multiple security solutions. Automation begets flexibility: developing alternative solutions manually and comparing them is not a feasible option. Similarly, application developers will use the tools to retrofit security to existing systems. They can apply the tools to remove a newly-discovered vulnerability, or just to upgrade their application.

The prediction that security-oriented program transformations will improve security engineering is speculative. It depends on tool vendors producing these tools, and developers adopting them. However, two factors indicate that program transformations can indeed be useful. First, we have applied proof-of-concept program transformation tools to remove real vulnerability from real systems; these tools work. The second factor is that these tools are similar to refactoring tools. Developers are already familiar with using refactoring tools as part of their IDEs. If the tool vendors incorporate these tools with IDEs, people are likely to adopt them.

### 1.1.3 Summary of Contributions

Our main work product is the catalog of program transformations that solve the most important security problems faced by today’s developers. These program transformations ease the task of retrofitting security to existing systems and make the whole process systematic. Our work also describes how to apply the security oriented program transformations, covering issues such as composition of security solutions to have defense in depth. Finally, the use of security-oriented program transformations improves the traditional security engineering.
1.2 Plan of Dissertation

Chapter 2 describes two case studies of software evolution to meet the changing security requirements. It will motivate the need for a new approach to security engineering. Chapter 3 surveys traditional security engineering approaches and identifies their shortcomings. Chapter 4 introduces security-oriented program transformations and illustrates how they can overcome the shortcomings. Then we give an overview of the program transformation catalog in chapter 5. The catalog is classified based on vulnerability classes. Chapters 6 through 11 describes each of the vulnerability classes and corresponding program transformations. These chapters describe the 37 program transformations in the catalog, how the transformations are classified, and how they can be applied together. Chapter 13 argues how the program transformations in our catalog solve the most important security problems. Chapter 14 concludes.
Chapter 2

Software Evolution Case Studies

This chapter tells two stories of software evolution. The first one is a study of the architecture of four mail transfer agents [93]: sendmail, qmail, Postfix, and MeTa1. These mail transfer agents are important checkpoints in the overall story of evolution of mail transfer agent architecture. They illustrate how the architecture has changed to meet the changing security requirements. The second story is about the evolution of Microsoft Word. This evolution is more user-driven: end users request new features in Word and developers add them. The story of evolution of Microsoft Word narrates how some of the design decisions sacrificed security in favor of other requirements, and how the developers had to catch up later.

There are two lessons to take from these case studies. First, they show that developers build their software applications with the set of security requirements they determine at the time an application is developed. Often, the security requirements are incomplete, and in the worst case, concerning legacy applications developed before the Internet era, security requirements are not even considered. As a result, developers have to continuously deal with changing security requirements. They cannot start developing from scratch when requirements change; they have to retrofit security to existing systems. The second lesson is the difficulty for developers when they try to retrofit security. It highlights the importance of getting the security requirements right when a software is designed. But since the requirements change anyway, it motivates new approaches to retrofitting security, which is the main focus of our research.

2.1 Evolution of Mail Transfer Agents

A mail transfer agent (MTA) [121] is a computer program that transfers electronic mail messages from one computer to another. MTAs are transparent to users. Normally, a user interacts with a mail user agent (MUA) [121] program, which uses an MTA to deliver email. The most popular protocol for mail delivery is the Simple Mail Transfer Protocol (SMTP) [160] [121], although it has been extended with Extended SMTP
ESMTP) [122]. The most popular Unix based MTAs are sendmail, qmail, Postfix, exim, etc.

Security is an important architectural requirement for an MTA. An MTA must keep email secure against casual attacks, and must not allow attackers to break into the system. The system must fail securely and should be recoverable to its original state, and it must also be able to track the cause of a security breach. An MTA must be reliable. It should never lose an email. An MTA must be efficient in both time and space, because it might process thousands of messages simultaneously. An MTA must have high availability. It must be able to cope with Denial of Service (DoS) attacks. Finally, an MTA must support existing mail transfer protocols, but it should be extensible for future protocols.

2.1.1 From sendmail to MeTa1

We have studied the architecture of sendmail, qmail, Postfix, and MeTa1. sendmail was the first standard MTA. It had a monolithic architecture and a lot of security problems. qmail was developed as a secure alternative to sendmail. Later, Postfix developers designed a secure MTA, but they also made it fast. MeTa1 is a recent attempt by sendmail developers to design a secure MTA; it has been designed from scratch. MeTa1 shows the influence of qmail and Postfix and is an example of the architectural lifecycle [168]: how “monolithic, insecure, unreliable and slow” can become “modular, secure, reliable and fast”. These four mail transfer agents illustrate the overall evolution: qmail was a reaction against sendmail to make it secure, Postfix was a deliberate attempt to better qmail, and MeTa1 was a response to its competitors.

Eric Allman released sendmail [186] with 4.1c BSD in 1980. The early sendmail architecture was ideal for its time, because the main requirement was flexibility with protocols and the architecture supported a lot of diverse protocols. Security and reliability requirements were not part of its original requirements.

The growth of the Internet made security and reliability important requirements for an MTA; this exposed the weakness of sendmail architecture. The first sendmail vulnerability was exploited by the Morris worm [63]. Since then, there have been 23 vulnerabilities reported for different versions of sendmail [93] [34]. For each instance, sendmail developers came up with a patch to solve the problem that has been reported.

As sendmail developers were struggling to make their system secure, competitors came up with better and more secure mail transfer agent architectures. The pioneering secure mail transfer agent was qmail [167]. Daniel Bernstein designed qmail as a replacement for sendmail with improved security; it does
not have any reported incidents of security vulnerability. Postfix [161], another competitor of sendmail, was designed in 1998 by Wietse Zweitez Venema. Its architecture was designed as much for performance as for security [93], but it also has a clean record of security. Both qmail and Postfix influenced the architecture of other MTAs that followed. In fact, the new generation of sendmail, MeTa1 [10], is not an evolution of previous versions of sendmail (sendmail version 8). Instead, MeTa1 follows the architecture of Postfix very closely.

Security is the main driving force behind the evolution of mail transfer agent architecture [93]. When security requirements changed, developers wrote new mail transfer agents. There were many mail transfer agents, but none of them shared any code. The developers were not retrofitting security; in fact, the case study highlights sendmail developers’ tribulations when they tried to retrofit security. Eventually, they went back to the design board and redesigned an MTA from scratch, reusing design decisions from qmail and Postfix. The next section describes some of these design decisions.

2.1.2 Changes in Mail Transfer Agent Architecture

An MTA accepts email from a local user and delivers it to a local mailbox or to a remote MTA. It also accepts email from another MTA. Thus, an MTA has two ways to accept input and two ways to deliver output, as shown in figure 2.1. An MTA collects incoming mail and processes it periodically using a mail queue, also shown in the figure.

sendmail has a flexible but monolithic architecture. It runs as one big process with root privilege\(^1\); the monolithic process receives, stores and delivers email. Some of these tasks are non-critical and do

\(^1\)Starting with version 8.12.0 released in September 2001, sendmail no longer installs as setuid root by default.
not require root privilege; users typically configure sendmail to run with a higher privilege and lower the privilege when it performs a non-critical task. But configuring sendmail is very complex. This complexity arises from the flexibility of sendmail\(^2\) (its support of multiple protocols and diverse services).

Most of the sendmail vulnerabilities originate from mistakes made in configuring the MTA. The mistakes are more severe because if one malicious user gets control of the monolithic sendmail process, he can use the root privilege to get control of the whole system.

qmail shows how to modularize the software architecture; modularity in its architecture begets simplicity, security and reliability. qmail’s designer, Daniel Bernstein, partitions\(^{[210]}\) the monolithic process into several processes. For example, the process that receives email from remote users, i.e., the SMTP server, runs in a separate address space from the rest of the MTA. This process submits incoming email to a process that handles the mail queue. The mail queue management process runs in a separate address space, and with a separate privilege level. Thus, even if an attacker discovers a vulnerability in the SMTP server, he or she can do little damage to qmail, because the exploit cannot go beyond the SMTP server process.

Partitioning is only one of the security solutions adopted in the qmail architecture. qmail has many other security solutions at different layers; it is a classic example of following the defense in depth principle\(^{[210]}\). To prevent unvalidated input vulnerabilities (such as buffer overflow and format string attacks), qmail replaces the C library for buffer handling with a safe library. The safe library keeps length and memory allocation information with each data buffer and uses it to decide whether a buffer operation is safe or not.

Another crucial aspect of the MTA architecture is reliable mail storage management. An MTA should never loose an email; even during failure, an MTA should recover gracefully without loosing any email. sendmail mailboxes and mail queue follow the classic single-file-per-mailbox style. This creates problems when multiple processes try to write to the file. qmail’s mailboxes (and mail queue) are implemented as directories; mail messages are written as separate files. The mail delivery process is implemented as a state machine. Each of the states are persisted in the file system. When qmail crashes, it can gracefully restart from the last stored state. This ensures the reliable management of mail messages.

\(^2\)Michael Zalewski reported one such vulnerability in sendmail 8.12.0 in 2001\(^{[33]}\). When it is processing untrusted information, sendmail is supposed to drop all extra privileges and continue to run at user level. Because of a programming error, sendmail 8.12.0 code fails to drop extra group privileges completely in new setgid conditions, leaving the saved gid value untouched. By calling the \texttt{setregid} function, an attacker can regain dropped privileges. This is possible due to several bugs in the configuration file parser. The problem was solved in sendmail 8.12.1.
Postfix shows how to increase performance without compromising security. The Postfix architecture is similar to the architecture of qmail. The main difference between qmail and Postfix is that Postfix considers performance as one of its quality requirements. Performance is not the main requirement in qmail, although the simplicity of its architecture also makes it fast. On the contrary, most of the design decisions of Postfix are influenced by performance or a performance-security tradeoff. Its design decisions illustrate how performance and security can both be achieved in an architecture\(^3\).

Eric Allman, the author of sendmail, planned a new version of sendmail with a modified architecture and fresh architectural perspectives to satisfy emerging architectural requirements like security and reliability [52]. This new generation of sendmail is MeTa1. MeTa1 closely follows the architecture of Postfix [10]. It does not follow the monolithic architecture: instead, its processes are partitioned according to their functionality.

This case study highlights the difficulty faced by sendmail developers as they attempted to retrofit security. Our research explores approaches to make this task easier.

### 2.2 Evolution of Microsoft Word

Microsoft Word was first released in 1983 under the name Multi-Tool Word [4]. It was about 27000 lines of code written for the Xenix systems. The DOS version for the IBM PC was released in the same year. Several versions of Microsoft Word was released in the 80s for various platforms, but only the versions for Macintosh were successful. Among DOS based word processors, it trailed behind the popular Wordstar and WordPerfect applications [162].

The fortunes started to change with the release of Word for Windows, which was written from scratch for Windows 3.0 [162]. Subsequent releases gained popularity as the popularity of Windows grew. Microsoft Word’s competitors refused to embrace a Windows based version until it was too late. By the time Windows 95 was released, Word 95 had become the dominant word processor. It was about 2 million lines of code. Since then, it has continued to grow as developers added more and more features to the existing codebase. In 2004, Word had about 1600 distinct commands, with roughly 50% of the commands remaining enabled.

\(^3\)Postfix preforks processes and reuses them by maintaining a process pool. This improves its performance. However, reusing the processes for a long time could open up a security vulnerability. An attacker can compromise a process and use it to compromise other processes. Postfix prevents it by killing processes after a predefined period and forking new processes in the process pool. This hinders performance; but improves the security. This is an example of Secure Preforking [84] pattern. The pattern applies to applications that maintain a resource pool, e.g., Apache server follows this pattern.
at any given time [182]. With just three steps, the possible combinations of code execution paths exceeded 500 million. Conservative estimates of the current size of Word codebase is about 30 million lines of code.

The native file format of Microsoft Word has changed several times; each time the developers modified the source code so that it understands the new file format. Word 97 to Word 2003 follows a binary file format [136] that implements OLE (Object Linking and Embedding) structured storage. Word 2007 uses Office Open XML [61], an XML-like standard, as its default format as well as supporting the binary format for backward compatibility.

Microsoft Word releases have exhibited two major types of vulnerabilities [34]: buffer overflow vulnerability originating from the use of unsafe buffer handling functions, and data injection vulnerability originating from its macro processing feature. Out of the 12 reported vulnerabilities [34] of Word 97, 6 are macro processing vulnerabilities, while 3 are buffer overflows. Word 2000 has 46 reported vulnerabilities [34]; out of that, 31 macro processing vulnerabilities and 8 buffer overflows. The most recent version of Microsoft Word, Word 2007, has 11 reported vulnerabilities [34]; 8 macro processing vulnerabilities and 2 buffer overflows.

Macro processing vulnerabilities originate from the design decision to mix code with data. Microsoft Word documents support embedded macros or even programs. Originally WordBasic was used, but Visual BASIC for Applications (VBA) has been the preferred macro language since Word 97 [19]. Macros run automatically when a document is opened. This gives an attacker the opportunity to write macro viruses and spread them using Word documents.

The design decision to allow code to be passed along with data is the root cause of all data injection attacks. In Microsoft Word, the design decision was to allow code to be passed, expecting that the code would be benign. The intent was to make documents portable across various platforms; security was clearly not considered. This vulnerability is exploited by attackers, who replaced the regular macro commands with the same name and ran those malicious commands in place of the originals.

The first reported macro virus is the Concept virus [19], released in 1995. The attacker, in this case, crafts a Word document overwriting the AutoOpen macro. Usually Microsoft Word calls an AutoOpen macro every time it opens a document. When a user opens an infected Word document, the malicious

---

4 Except for the format change from Word for DOS to Word for Windows. In that case, the developers wrote Word for Windows from scratch.
version of AutoOpen infects Normal.dot file\(^5\) by adding four new macros: AAAZAO, AAAZFS, PayLoad and FileSaveAs\(^6\). The user’s system is infected from that point onward. Whenever the user tries to save a file, the malicious FileSaveAs macro in the normal.dot file is executed. This creates a malicious AutoOpen macro in the new document, and copies the content of AAAZAO into it. Concept, as its name implies, is a proof of concept macro virus that only spreads and infects new machines; it does not have a malicious payload. However, its famous successors Melissa and VBS.LoveLetter makes several malicious changes to the infected system. Melissa [41] caused millions of dollars of damage. VBS.LoveLetter was more malicious: it infected 10% of all computers connected to the Internet causing $5.5 billion in damage [76].

Buffer overflow vulnerabilities in Microsoft Word originate from insufficiently checking input buffers before they are used. Writing rigorous checks for all instances in a codebase as large as 30 million lines is prone to human errors. Attackers expose these corner cases that have escaped rigorous testing. Every time a vulnerability is reported, developers fix that particular instance of buffer overflow, but other cases of buffer overflow might still remain for future exposure.

The Microsoft Word case study provides two lessons. First, it shows that security requirements are often downplayed when a feature is added to an application. When developers added macros, their intent was to add support so that repetitive document production tasks could be automated [123]. Security implications of macros were not considered. Microsoft’s response to macro viruses is to set a tighter security policy by default, so that macros are not automatically executed. This is not a direct fix, but the approach has managed to reduce the risk from macro-based viruses. The second lesson is the difficulty in managing the huge code base. It is hard to maintain the huge code base and retrofit buffer overflow prevention mechanism to it.

### 2.3 Lessons Learned

Security requirements change. Even systems that are built to be secure, has to adapt to meet with the changing security requirements. The case studies illustrate how the change of security requirements impact a software application. sendmail was an architectural marvel; its developer elegantly designed based on a set of requirements. On the other hand, Microsoft Word’s macro feature was a design decision intended to make

---

\(^5\)Normal.dot file is the master template for all Word files.
\(^6\)AAAZAO contains the payload that would be copied to AutoOpen macro when a new file is infected. AAAZFS and FileSaveAs are identical. AutoOpen macro searches for the AAAZFS macro to check whether a system is already infected. PayLoad macro is never executed. It contains a comment only: “That’s enough to prove my point.”
Word documents portable across various platforms. Clearly these design decisions were not motivated by security. Design decisions such as these commonly riddle application developers and force them to attempt to retrofit security.

These case studies also highlight the struggle of system developers to retrofit security. sendmail developers, following a penetrate and patch [170] paradigm, eventually decided to completely redesign it. Microsoft Word developers have also redesigned their application from scratch, when it was developed for the Windows environment. But this effort does not scale for the current version. Since developers will have to continue retrofitting security, systematic, general-purpose support for retrofitting security is valuable.

The next chapter further motivates security-oriented program transformations by describing why the traditional approaches of security engineering need to be improved.
“Programs, like people, get old” [155]. Aging occurs in applications with bad design; also, in successful applications with good design. Good design and software engineering only delays the aging process. Similarly, a sound security engineering methodology aims to ensure that the value of an application is not diminished by its security vulnerabilities.

Security engineering activities are prominent not only during the software development process, but also after the release of software. The pre-release phase revolves around sound requirement engineering to create a threat model and designing countermeasures; we will refer to these activities as ‘Secure Software Development’ activities. The post-release phase involves writing and distributing patches for fixing newly discovered vulnerabilities or upgrading routinely; we will refer to these activities as ‘Secure Software Maintenance’ activities.

Both approaches have inherent problems that contribute to software aging. Secure software development approaches recommend careful upfront design. Developers focus on getting the security design right, the first time they build an application. Integrating security upfront in the design phase is imperative. However, this approach produces many insecure applications because developers fail to understand the security requirements. Moreover, security requirements change frequently and new classes of security vulnerabilities appear. These changes occur when an application is being developed, as well as after an application has been released. Each time developers have to find a way to adapt an application to meet the changing requirements. On the other hand, secure software maintenance approaches are based on writing patches. Writing patches is an ad hoc, manual activity, that is error prone. Hence, a patch often fails to remove a target vulnerability, or it introduces new vulnerabilities.

Consider the story of sendmail, described in the chapter 2. sendmail was developed before the Internet; security and reliability requirements were not considered by its developer. When sendmail’s vulnerabilities were discovered later, its maintainers patched the affected parts. The maintainers only attempted to fix the
vulnerability that had been reported, instead of attempting to fix the overall security problems of sendmail. As a result, the same vulnerability reappeared in sendmail. For example, buffer overflow vulnerability had been reported five times in different parts of sendmail [93].

This chapter surveys secure software development and secure software maintenance approaches and identifies why the approaches fail to delay software aging. Our discussion unfolds the main problem of traditional approaches: they do not describe a systematic method to adapt software to meet changing security requirements. This motivates our research on a systematic methodology to introduce/retrofit security to applications, aka security-oriented program transformations.

3.1 Secure Software Development Approaches

The aim of a secure software development approach is to formally describe the development activities and the standards to which the end products should comply. Initial efforts to define a secure software development approach resulted in standards to assess whether a software engineering process sufficiently considers security and whether a developed application meets security requirements. This approach has been replaced with a process based approach, which attempts to integrate security at every stage of software development—project definition, requirement analysis, design, development, testing, and release. The process based approach does not define new stages of software development; it adds new tasks (e.g., create a threat model during requirement analysis and design) and new role players (e.g., security architects) to the existing stages of software development process. A reactive effort to define a secure software development approach comes from the agile community: the approach does not rely on separate stages of software/security development. Instead, the agile way is to let the security requirements evolve throughout the development stage and to allow these requirements to guide the development process.

Next, we will describe these three efforts: standards based approaches, software process based approaches based approaches and agile approaches.
One thing is common in all the software development approaches. They depend on gathering security requirements efficiently. However, all of the approaches acknowledge that software will continue to evolve even after release. Hence they all rely on 1 software maintenance, typically using patches. We will describe patch based software maintenance approaches in section 3.3.

3.1.1 Defining Standards for Secure Software Development

Initially, researchers and practitioners defined standards to assess the software development process and the end product. These standards classify software applications among different security levels, but they do not describe a security engineering process to follow to get to any level.

Standards for End Product. The first standard for secure software is the Trusted Computer System Evaluation Criteria (TCSEC) [59] formulated by the United States Government Department of Defense (DoD) in 1985. It provides guidelines to select computer systems that handle sensitive information. TCSEC defines four levels of trust that a system can be in. Similar effort in Europe produced Information Technology Security Evaluation Criteria (ITSEC) [50] in 1991. The levels of ITSEC has loose semantic equivalence with the the levels of TCSEC. Efforts to combine these two standards into a common standard first resulted in Canadian Trusted Computer Product Evaluation Criteria (CTCPEC) [37] and then in Common Criteria for Information Technology Security Evaluation (Common Criteria or CC) [111]. The levels of security in Common Criteria map to the security levels of previous standards but the evaluation process is different.

Common Criteria determines security by evaluating two things. First, it identifies a set of protection profiles (PP) that include the desired security requirements of users performing different tasks. Each PP encapsulates a class of security devices, e.g., the PP for a firewall. Secure systems and software are evaluated to identify whether they fulfill one or more protection profiles. Second, Common Criteria defines security targets (ST), i.e., security properties of the target software or system. The security requirements for each target software or system is different; two network firewalls might have completely different security targets.

Standards for Software Development Process. There are standards to evaluate security engineering processes. Capability Maturity Models (CMM) [156] measure the efficiency of a software engineering process. Among the CMM’s, Systems Security Engineering Capability Maturity Model (SSE-CMM) [134] provides

1Many approaches also integrate maintenance in their definition.
a reference model to evaluate a security engineering process. The SSE-CMM also describes the desirable goals of an organization’s security engineering process, goals such as providing information about the security risks and corresponding security needs, transforming security needs into security guidance to be followed during the design and development process, and establishing assurance in the effectiveness of security mechanisms.

SSE-CMM does not define a full-fledged security engineering process; it identifies 5 important security-related subtasks in the software development life cycle. They are, 1) specify security needs, 2) provide security input, 3) monitor security posture, 4) administer security controls, and 5) coordinate security. These tasks extend the original stages of a software development life cycle\(^2\).

The next section describes the approaches that aim to redefine the software development process.

### 3.1.2 Integrating Security During Software Development Lifecycle

The push for a formalized security engineering process started in Microsoft following its Trustworthy Computing initiative [144]. The process that originated from the initiative, titled Secure by Design, Secure by Default, Secure in Deployment and Communications (SD\(^3\)+C) [144], is a guideline for software designers and developers. According to it, software should be secure by design; the requirement analysis, design and implementation phases should enumerate types of attacks and design countermeasures following certain guidelines. Security must be considered in the requirement analysis and design phase. Finally, SD\(^3\)+C guides end users and administrators to securely deploy and use software. The communication aspect of SD\(^3\)+C guides software maintenance, e.g., how to deal with a discovery of product vulnerabilities, how to create and deploy patches, etc.

The standard that followed SD\(^3\)+C, Microsoft’s security development life cycle (SDL) [126], is a detailed security engineering process. It defines 6 phases for secure software development: requirements phase, design phase, implementation phase, verification phase, release phase, and support and servicing phase. The phases of SDL follow the phases of software development process, with additional security-related tasks in each phase.

SDL also defines new role players in an organization. To successfully follow SDL, an organization should have a central security team that will work with the product development team. The product team

\(^2\)The original stages of software development life cycle are concept stage, development stage, production stage, utilization stage, support stage and retirement stage [106].
appoints a security advisor who acts as a bridge between the two teams. The security advisor reviews security plans, makes recommendations, and advises on the security milestones for the product. In the first two phases of SDL, the advisor prepares a threat model at a component-by-component level and designs the countermeasures. The threat model drives the subsequent phases of software development. It sets the standards for coding, testing and integration, guides the code review process, and sets the security acceptance criteria of a product to be released. Whenever the threat model changes, the security design and development practices are changed accordingly.

SDL has gone through several stages of modification. Current version of SDL [137] breaks the security engineering process into 14 phases. The phases are the same as those of previous versions, only some of the phases are further broken down. However, there is one important difference with the previous versions of SDL. The current version emphasizes preparing a team for secure software development, e.g., training all members of software development team on secure design principles, threat modeling, secure coding practices, security testing methods and privacy enhancing technologies. The preparedness of the team contributes to a comprehensive threat model that guides the subsequent software development phases.

Team Software Process (TSP) [103] of the Software Engineering Institute (SEI) provides a framework for applying software engineering principles at the team and individual level. TSP for secure software development (TSP-Secure) defines a method that can predictably produce secure software. TSP-Secure addresses secure software development in 3 ways: efficient planning and scheduling, quality control, and security training. Teams using TSP-Secure build their own plans during a project launch. Typical tasks included in the plan are identifying security risks, eliciting and defining security requirements, secure design and code reviews, and use of static analysis tools, unit tests, and fuzz testing. The team is led by a security manager through different phases of development. The software product is tested in multiple phases to ensure high quality. Finally, TSP-Secure includes training for the actors in a software development process.

Another structured approach for moving security concerns into the early stages of the development lifecycle is the Comprehensive, Lightweight Application Security Process (CLASP) [150]. CLASP supports security-centric software development; it defines a set of 24 top level activities and resources that have to be integrated in the development process and the environment. It treats the security activities in great detail, and outlines the exact responsibility of different role players during each activity. CLASP activities are lightweight and are easier to integrate to the existing development process than SDL [81].
3.1.3 Being Agile

The recent trends for security engineering focus on incrementally building the security architecture. Agile security engineering approach advocates incremental threat modeling instead of the up front threat modeling favored by other approaches. Agile security engineering defines good enough security [17] instead of absolute security. Agile practices let the customers be in charge of defining the security requirements that are good enough for their product.

An example is extreme security engineering (XSE) [44]. During the planning game of XSE, the development team estimates each user story in terms of ideal development weeks. The customer then decides which subset of the stories are more important for the security of the product. Planning is not done in advance; developers and customers engage in the planning game just before each iteration. Other benefits of XSE, claimed by its proponents, come from small releases and testing. Small, frequent releases allow customers enough opportunities to review their definition of good enough security and reprioritize their user stories. Testing increases the confidence of both developers and customers as fluctuations happen. During the iterations, the security architecture grows hand-in-hand with the code.

3.2 Up Front Security Design

Most of the security engineering approaches, listed in the previous section, rely on a comprehensive threat model prepared up front to guide the rest of the development process. Threat modeling and rigorous requirements analysis have been explored by many researchers. Sindre and Opdahl [190] have extended UML use cases to identify and capture security threats in the requirement analysis phase; these are called abuse case. Xu and Pauli [219] have augmented UML sequence diagrams to model the threats. Haley et al. [95] have described a framework for eliciting and analyzing security requirements. It marries concepts from requirements engineering and security engineering, and defines a detailed process for determining security goals, converting them to security requirements, iteratively building a threat model, and verifying the model. Butler [36] has described a cost-benefit analysis approach to identify whether the security solutions in an application effectively covers the security requirements. Microsoft follows a threat modeling approach that is based on textual description and attack trees [199]. The approach relies on the STRIDE and DREAD model [199] as a guideline. STRIDE enumerates 6 types of security threats and DREAD defines 5 severity
levels of attacks. Together they are used to appraise security threats and their severity.

Agile practices critique the up-front threat modeling approach and distribute threat modeling throughout the entire software development life cycle. Agile security engineering practices rely on architectural risk analysis [77] and constant refactoring [11]. These practices do not eliminate threat modeling; a developer starts with a rough model for security goals, and then refines them at each iteration.

Regardless of the philosophy, security engineering approaches require the presence of a security architect in the design phase because “security cannot be added on, it must be designed from the beginning” [7]. It is imperative to design carefully based on a set of concrete security requirements (threat model) because adding security later is impossible or at least economically infeasible.

“Software security best practices leverage good software engineering practice and involve thinking about security early in the software life cycle, knowing and understanding common threats (including language based flaws and pitfalls), designing for security, and subjecting all software artifacts to thorough objective risk analysis and testing.” [131]

The rationale behind this up-front approach is that “it’s just plain easier to protect something that is defect-free than something riddled with vulnerabilities” [131]. Significant cost can be saved when security analysis and engineering approaches are introduced early in the development cycle.

“The return on investment ranges from 12 percent to 21 percent, with the highest rate of return occurring when analysis is performed during application design.” [102]

Security becomes prohibitively expensive to retrofit and a retrofitting approach breaks a lot of system properties [18]. Wäyrynen et. al. [218] suggests that security engineers should be part of an agile process from the start, even if they are only involved part time.

### 3.3 Secure Software Maintenance Approaches

Software development processes do not end with the release of the product; post-release software maintenance is an integral part of the development life-cycle. When new security vulnerabilities are discovered, software vendors write patches and try to distribute them before the vulnerability is widely exploited. Patches are used for many reasons, e.g., adding a new functionality, addressing a security flaw etc. We
will concentrate on security patches only\(^3\). Patch development is not different from secure software development; it is guided by the requirement that a security flaw in a part of the code has to be removed. The developer fixes the flaw, tests the solution, and releases the patch to end users.

Users of a software application should update their product when patches are released. But, failure to update is one of the most common issues identified by security professionals. It is difficult for system administrators to keep track of numerous patches coming out every day. Software maintenance standards recommended that organizations create a patch and vulnerability management group (PVG) [132]. The PVG is the central point of the vulnerability remediation effort. The standard for patch management [133] describes the tasks for PVG, from vulnerability identification to applying patches.

### 3.4 Aging and Secure Software Development Approaches

There are two distinct failures of software maintainers that contribute to software aging [155]: one is failure to meet changing requirements; the other is failure to make appropriate adaptation when a change is required. These failures are evident in secure software development and maintenance approaches.

Traditional approaches recommend careful upfront design. However, new security threats emerge after software release. Software architects fail to meet the changing requirements because:

**Waterfall Effect.** The first, and perhaps still ubiquitous, model for engineering software is the waterfall model [23]; everything has to be planned to perfection because there is no way of going back in the software development life cycle. Waterfall model has a lasting impression over all other software engineering processes that followed. On a positive note, subsequent process models [175, 24, 124, 47] improve the shortcomings identified in the waterfall approach. On the other hand, the specters of waterfall model remain in many aspects of software development processes, security engineering being the most notable (and relevant) example.

The security software development approaches embrace the waterfall approach because they depend on a sound threat model prepared up-front to guide subsequent phases. Up-front threat models are almost always inadequate because requirements are insufficiently understood at the start. Besides, new threats emerge during software development life cycle. Correspondingly threat models change, but the change requires a

\(^3\)We use the term security patch and patch interchangeably. Both refer to ad hoc patches that are prepared and distributed for making security updates.
huge investment of effort [102]. Agile security engineering processes recommend the use of refactoring at each iteration to make systems more conducive to retrofitting security [44, 11, 77], but practices show that results are not encouraging and retrofitting involves a lot of work [18].

Many new security threats emerge after software release. Threats in the post-release phase impact all software, both new systems that have been designed following a sound security engineering approach, and legacy systems that have not been designed with security in mind. According to the waterfall model, it is impossible to retrofit security in these cases unless the software is redesigned with a new threat model.

**Infeasible to Redesign.** Redesigning a large software system is infeasible. Even if redesign is an option, the effort cannot lock step with rapidly changing security requirements.

In the MTA case study, sendmail developers decided to redesign it from scratch in order to make it secure. But, sendmail could be redesigned because it was small (less than 100K lines of code). In spite of this, it took the developers almost 25 years before they started the redesign effort. In the Word case study, Microsoft developers redesigned it from scratch in order to port it to the Windows environment. But it does not scale for the current versions of Word, that are roughly 30 million lines of code. In fact, Microsoft tried and failed to redesign it from scratch in their Pyramid project⁴.

**Manual Change.** Regardless of the phase of software development, modifying for security is mostly a manual activity. Automated tools for security engineering are either non-existent or unpopular. Most of the available tools detect various types of security vulnerabilities in an application and leave it to the developers to write the fixes.

To understand whether developers use tools for retrofitting security, we selected a simple task that could be automated and surveyed how developers do it. We wanted to know how developers modify their applications to prevent buffer overflow vulnerabilities. One of the main reasons behind buffer overflow vulnerabilities is the use of unsafe buffer functions in C/C++ programs that do not check for bounds before writing to a buffer. There are opportunities to automate this task, but usually programmers manually inspect code to add bounds checking or replace unsafe libraries with safe libraries in order to prevent the vulnerability.

We asked the developers of the top 10 most active projects in sourceforge.net [191] about their devel-

---

³Pyramid project was an attempt to produce a cross-platform Word suitable for Windows and Macintosh environment. The project was abandoned when it was determined that it would take the development team too long to rewrite and then catch up with all the new capabilities that could have been added in the same time.
opment approach. 6 projects use C or C++ as one of the languages, 3 use PHP and 1 uses Java. In 5 out of 6 C/C++ projects, programmers initially used `strcpy` and `strcat` functions, but manually changed to safer C/C++ string libraries later. All of these changes were manually done by contributors, typically by `grep`-ping for unsafe functions and replacing them manually. Among the 5 projects, gimp-print, licq and pidgin maintainers replaced the unsafe functions after buffer overflow vulnerabilities were found, whereas Crystal Space 3D SDK and Ghostscript maintainers changed as routine maintenance. The remaining C/C++ project, numpy uses safe string functions in its original design.

In all the cases, developers used tools to find buffer overflow vulnerabilities, or find instances of unsafe string functions in source code. But making the change, either by writing tighter bounds checks or by replacing unsafe functions, is a manual activity.

For large projects, this requires a lot of manual effort; consequently, contributors of large projects only change instances of functions that they judge unsafe. For example, Ghostscript (about 350 KLOC) is a medium size project, but only a small part of its codebase has safe functions in place of the unsafe functions.

### 3.5 Aging and Secure Software Maintenance Approaches

To deal with threats emerging in the post-release phase, software maintainers follow an approach based on security patches. Patches are written in an ad hoc manner to fix a newly discovered vulnerability. Patch-based security engineering fails to make appropriate adaptation for many reasons.

Patches are written to fix a newly discovered vulnerability, but they have a limited scope. The following problems frustrate a patch-based security engineering approach and hasten software aging.

**Limited Scope.** Software maintainers follow the ‘penetrate and patch’ paradigm [170]: when a vulnerability is reported, they write a security patch for the software. A patch has a limited scope; it only removes the vulnerability at a fixed set of points. The same problem might remain in other parts of the program.

sendmail had five buffer overflow vulnerabilities reported at different times [93]. Each time the maintainers responded with a patch, only to let the vulnerability resurface in another part of the source code.

**Hasty Surgery.** Security patches are written when a new vulnerability is reported. Patch developers work under pressure of fixing the vulnerability in a short time. This might lead to buggy patches. For example,
Microsoft security bulletin MS06-42 contained a patch for Internet Explorer version 6 service pack 1, but within a few days eEye Digital Security reported a critical new security bug in the patch itself [139].

Another symptom of hasty surgery is the failure to understand the problem. The resulting security patches might fail to solve the problem, or they might solve the problem incompletely. Adobe released a security patch to overcome a vulnerability [31] on October 22 2007, only to realize that the patch did not solve the problem [169]. Microsoft security bulletin MS05-018 contained a fix that partially solved the vulnerability [39].

**Drowning Effect.** In 2007, CERT coordination center (CERT/CC) [40] cataloged 7236 security vulnerabilities. Although many security patches fix multiple vulnerabilities, the large number of vulnerabilities correlate to a large number of patches. It is difficult for an organization to keep track of all the patches required to safeguard their systems. This leads to many security exploits. For example, Microsoft developed the security patch [140] for the vulnerability exploited by the Slammer virus 6 months before the attack, yet the virus caused network outages all over the world.

### 3.6 Summary

This chapter describes security design and maintenance approaches in practice and identifies their shortcomings. Software must be continually changed to meet new threats, but the current trend is to make ad hoc, manual changes. These manual changes are inefficient and error prone. In fact, the problems identified in this chapter are general problems that riddle any developer when he/she attempts to manually make changes. The next chapter introduces security-oriented program transformations, that allow a developer to make changes in a more systematic manner.
Chapter 4

Security-oriented Program Transformations

A program transformation is a function that maps input programs to output programs. Transformations can be divided into two types: translation and rephrasing\(^1\) [163]. In a translation scenario, a program is transformed from a source language into a program in a different target language, e.g. compiling a C program to object code. On the other hand, a rephrasing transformation modifies an input program to a different program in the same language, e.g. refactoring to make a program more understandable. In both cases, a program transformation modifies program structure and retains program semantics.

In this chapter, we will give an overview of security-oriented program transformations, i.e. program transformations that add security properties to existing programs. We will introduce security-oriented program transformations, describe how program transformations can be applied to make programs secure and discuss issues of applying security-oriented program transformations in practice.

4.1 Security-oriented Program Transformations

A security-oriented program transformation [89] is a function that maps programs to security-augmented programs.

Consider a program transformation that replaces some unsafe library functions in a program with safe alternatives. Such a program transformation can be used to remove the use of unsafe library functions and therefore the vulnerabilities originating from the use of these unsafe functions. An example application is to replace the instances of `strcat` and `strcpy` functions in C programs to prevent buffer overflow vulnerability.

Library replacement is a program transformation that applies at the program level. Other program transformations focus on the application architecture. For example, a partitioning transformation distributes

\(^1\)It is possible to divide program transformations in various ways. Here we have distinguished them based on the source and target language.
program components so that each of them runs with separate privilege level. This transformation changes
the components and their connectors.

Both partitioning and library replacement transformations can be viewed as a function that takes input
programs and produces output programs. Figure 4.1 shows a schematic diagram of such a function. Figure
4.1 also raises several concerns. What are the inputs of a security-oriented program transformation? What
is the specification for a transformation? What type of change is made by a security-oriented program
transformation? How many security-oriented program transformations are there? Finally, what do we mean
by improved security in the output programs?

rephrase an input program, i.e. the input programs and the target programs are of the same language.
However, program transformations include both source to source and binary to binary transformations.

Security-oriented program transformations are general-purpose. But, no program transformation will
work with every program (except the identity transformation); they usually expect a certain programming
language or a certain platform and apply to programs written in that language and/or platform. Different
programming languages and platforms will have different sets of transformations. For example, a safe li-
brary replacement transformation to prevent buffer overflow vulnerability is only applicable to C programs,
since buffer overflow vulnerability is C specific. Based on the implementation mechanism, the transforma-
tion can apply to C source code or binary code. On the other hand, a partitioning transformation can be
applicable to programs written in C, Java or other languages. But, an implementation of partitioning will
work with only one language.

A variant of a library replacement transformation can remove another type of vulnerability in a different language. All library
replacement techniques are described as one program transformation technique.

---

29
Specification for a Security-oriented Program Transformation. All types of program transformations require user specification. A user applying a refactoring transformation needs to specify where to make the change, e.g. which variable to rename, which method to extract etc. Even an automated transformation such as compilation requires user specification in the form of makefiles and build configurations. Security-oriented program transformations can be interactive similar to refactorings, or more automated similar to compilers. Nevertheless, they need some form of user specification.

Some program transformations such as a library replacement transformation are completely automated, but others are only semi-automated because they require some steps to be performed manually. A developer guides how a program transformation is applied by regulating the automated parts. A developer’s specification is the *policy*. The tools implement the *mechanism*; they automatically execute structural changes.

To use a security-oriented program transformation, a developer has to follow three steps.

1. Identify the program points where to apply a transformation.

2. Determine which transformation to apply.

3. Use a tool to automatically transform the program.

The first two tasks are manual; a developer identifies where to apply a transformation and which transformation to apply. Usually, a developer supplies these parameters to a program transformation with a manual specification.

The specification for a safe library replacement transformation is the choice of unsafe libraries (e.g. *strcat* and *strcpy*), and their safe alternatives (e.g. *g_strlcat* and *g_strlcpy* from the glib library). On the other hand, a large transformation such as partitioning requires a lot of specification. A partitioning transformation needs to be told which parts of the program go into each partition. A developer manually specifies these.

With this separation, program transformations can automate new security protections without encoding deep understanding of program behavior in the transformation. This makes the implementation of the transformations feasible. Programmers still require skill and domain knowledge to apply security-oriented program transformations, but like refactoring tools the program transformations make changes faster and more reliably. Since a key part of using a program transformation is choosing its parameters, when we describe a program transformation we will carefully describe its parameters.
The parameters of a security-oriented program transformation usually refer to parts of a program, e.g. functions, variables, classes etc. If these parameters are invalid, a security-oriented program transformation should signal an error.

**Structural Change made by a Security-oriented Program Transformation.** We restrict security-oriented program transformations to make structural changes similar to refactorings; structural changes such as replacing method calls, composing classes, refactoring to patterns, distributing artifacts, etc. This eases the cost of automation, because the corresponding tools do not need to have deep understanding of program behavior. A user provides the specification and a transformation tool performs the mechanical change.

Security-oriented program transformations, especially the ones that make architectural changes, are larger than typical refactorings. They can be compared to large refactorings comprising of many steps. These program transformations are interactive: some of these steps can be supported by tools while other steps can be manual. Section 13.2 discusses program transformations and the feasibility of building tools for program transformations.

The change, made by a program transformation, can be categorized by two issues: the scope of change and the mechanism of change. A security-oriented program transformation is platform specific and language specific. The same program transformation can make different types of changes in different contexts. For example, a partitioning transformation for a C program splits a process so that each partitioned process runs in its own address space. On the other hand, a partitioning transformation for a Java program distributes classes so that they can run in separate virtual machines. The mechanism of a partitioning tool for a C program is different from that for a Java program. Consequently, if an automated tool is built to implement the transformation for a C program, it will not work with a Java program. However, it should be general-purpose in the sense that a developer should be able to apply the transformation on any C program. Compare this to a patch program that uses diff outputs to transform a program. It is designed to work with a single version of a program. If diffs are considered, anything is a program transformation. But security-oriented program transformations are more general.

The impact of change made by a security-oriented program transformation distinguishes it from refactorings. Security-oriented program transformations are not behavior-preserving the way refactorings are. A security-oriented program transformation preserves the desired behavior and fixes the undesired behavior caused by a security vulnerability. Our program transformations are behavior-preserving when the system
is used correctly; they preserve good path behavior. Only attackers see the change in behavior, because
security-oriented program transformations eliminate the source of vulnerabilities the attackers want to ex-
plot. For example, the mechanism of a library replacement transformation is dependent on program flow
analysis, but the change that is made is structural. Nevertheless, the behavior of the program will change
after the functions are replaced. An attacker who was previously able to overflow a buffer in a program
would not be able to do so in the modified program. So, the behavior of the original program would be
different for him. For abiding users, the good path behavior of the program should remain the same.

Each program transformation is different; so is the implementation mechanism of a transformation tool.
We have written proof-of-concept tools in Perl. Though it works, Perl is not the best implementation option.
Special purpose languages such as TXL [51] and Stratego [211] are better; implementation approaches such
as aspect oriented programming can be an option [60]. Another option for building program transformation
tools is to use language infrastructures such as Eclipse CDT and JDT. The purpose of this dissertation is
not to figure out the best way to implement program transformations, it is instead to show that automated
program transformations can add security to existing software.

Number of Security-oriented Program Transformations. We have compiled a catalog of candidate se-
curity solutions that can be described as security-oriented program transformations. Currently, the catalog
contains 37 security-oriented program transformations. The catalog was created by surveying vulnerability
trend reports (e.g. [45, 207]) to identify relevant security problems, and using our catalog of security pat-
terns [92] to identify solutions for these problems. Chapter 5 describes how the program transformation
catalog was derived.

transformations introduce security solutions that developers otherwise will introduce manually. But, adopt-
ing a single security solution cannot secure a program; one must have defense in depth [210]. Defense
in depth principle suggests that each component of the information processing system must have its own
protection mechanisms. This advocates security solutions applied at different components, thus creating
defense in many layers. The security solutions can be isolated and target different vulnerabilities, or they
can be overlapped and target the same vulnerability. Using a defence in depth strategy, should one defensive
measure fail there are other defensive measures in place that continue to provide protection.
Composing multiple security solutions at different layers hardens a program. Each security-oriented program transformation contributes to an overall program hardening transformation. Security-oriented program transformations should be composable so that multiple transformations can be applied to a program; therefore we focus on the ones that map programs in a language to other programs in that same language.

By applying a library replacement transformation to replace `strcpy` and `strcat` functions, a developer can remove the potential buffer overflow vulnerabilities originating from the use of these functions. But buffer overflow vulnerability also originates from other sources: use of other unsafe functions (e.g. `memcpy`, `gets` etc), direct use of pointers, and integer overflow vulnerabilities. Other program transformations should be applied to remove these vulnerabilities. On top of all these, a C program can be partitioned so that the compartments run with the least privilege level. This does not directly remove buffer overflow vulnerabilities, but it makes buffer overflow vulnerabilities harder to exploit. Together, these transformations harden a program to secure it against buffer overflow attacks.

4.2 Related Work

Security-oriented program transformations bridge two domains: security patterns and tools for automating software redesign. Our work is based on the continuously growing body of security patterns [220] [21] [194] [184] [101] that document best practices in making systems secure. Most of the security-oriented program transformations in the catalog originate from our work on building a comprehensive catalog of all published security patterns [92, 91]. Today, security patterns are not as easy to use as other software patterns, because they do not offer precise advice. Instead, they provide abstract, general guidance that can be turned into complete solutions only by experienced security engineers. Security-oriented program transformations provide a practical mechanism of applying security patterns to improve security. The next chapter describes in detail how the security-oriented program transformation catalog has been derived from security patterns.

However, our work is related to software patterns in general, not just security patterns. The transformations also incorporate patterns from the fault tolerance domain [98]. The relationship between security-oriented program transformations and security patterns is of the same type as between the ‘Refactoring to patterns’ [116] book and the design patterns [74]. Refactoring to patterns describes, step by step, how to introduce design patterns into the code by applying a refactoring or a code edit at each step.
Throughout this dissertation, we will be comparing security-oriented program transformations and the ways they can be automated to refactoring [71]. Refactoring improves the quality of design and code by making changes that do not affect the observable behavior. This property, called behavior preservation, is not a part of security-oriented program transformations, because our goal is to improve the external behavior of the system by making it more secure. But in all the other aspects, this work parallels that of refactoring.

As a technique for describing how to modify software design and code, security-oriented program transformations are similar to other software documentation techniques. Design fragments [67] is a technique for describing how to use a software framework. It identifies the constraints that a framework user needs to meet in order to use it. These specifications are stored in the source code, as annotations.

Spitznagel et.al. [193] describes an approach to systematically augment basic connector types to produce more complicated connector types. They present a catalog of 8 connector transformations. Their approach influences ours in two ways. First, it describes a systematic methodology to derive a catalog of transformations. It derives a few very general transformations, whereas our approach derives a lot of fairly specific transformations. Although the approaches appear complementary, there are fundamental similarities in them. We compare the approaches in section 5.1.

The second influence of Spitznagel et.al.’s [193] work is on the implementation of our program transformations. Security-oriented program transformations embody various strategies: add and distribute components, modify connectors and control flow, etc. The steps that involve connector transformations are in fact one of the six connector transformations (or a combination of them). Section 13.2 describes how Spitznagel’s connector transformations can be used to implement security-oriented program transformations.

Applying program transformations to improve security of systems has only been studied in a few contexts, e.g. program partitioning [119, 28], buffer overflow prevention [56], etc. Recent work also explores applying genetic programming to mutate a program to create automatic patches [216], or use program profiles to find deviation and generate patches [159], but there scopes are limited. Each program transformation in our catalog narrates the corresponding attempts of applying program transformation techniques. Our description of security-oriented program transformations also include the general solution techniques that solve a security problem and how they can be automated. Hence the related work of security-oriented program transformations extends beyond this section into the description of each transformation.
Possibly the easiest way to implement some of these transformations in tools is to use aspects [117]. Logging and weaving in method calls to new components are the poster children of AOP. So transformations that add a module, such as the Add * transformations, can be implemented with AOP relatively easily. However, there are other transformations for which AOP does not offer immediate solutions. Simple scripts or transformations based on abstract syntax tree manipulation are other implementation alternatives for such tools. We will cover implementation techniques when describing the program transformations.

4.3 Security Engineering Using Program Transformations

Security-oriented program transformations can make it easier for programmers to keep up with security threats. They will be an integral part of developing and maintaining secure software. Automated security-oriented program transformations aim to have a greater coverage than a manually-written fix; in the long run, they will be cheaper to apply.

In order to be useful, developers should have tools available that they can use to apply the transformations. But who will build tools for security-oriented program transformations? In general, application developers will find it prohibitive to develop these tools themselves. On the other hand, it is more feasible for platform vendors and vendors of language IDEs to build these tools. Platform vendors can build general purpose tools fit for their platform. Language IDEs can have program transformations listed in their refactoring catalog. Platform vendors who develop program transformation tools will be called transformation tool vendors.

There are three different roles: end users, application developers and transformation tool vendors. End users of an application are interested in using secure software; they will apply the security updates periodically. Application developers design and develop secure applications; when new security vulnerabilities are reported or security requirements change, they will be interested in modifying their product and distributing security updates. Our work shows that application developers can have these program transformation tools at their disposal, tools that will relieve them from making manual changes. But the tools would usually not be created by application programmers. It will be the transformation tool vendors, who will develop general purpose tools for applying security solutions.

However, some program transformations may be application specific; application developers will have to write customized tools for these transformations. In this case, application developers also play the role of
transformation tool vendors. Suppose a program transformation tool is available that applies input validation policies to prevent SQL injection attacks. A generic tool includes a library of policies that prevent various types of SQL attack vectors. Developers can extend the generic tool to include additional organization-specific and domain-specific policies.

Having security-oriented program transformations as tools will aid both the software development and maintenance phase. It will enable application developers to make changes easily; they will be able to feasibly test competing security solutions and analyze the trade-offs.

On the other hand, when a developer needs to modify an application, two things can happen.

1. Program transformation tools may be available. Application developers will use them to generate a security-upgraded program.

2. Program transformation tools may be unavailable. At any time, a system will provide only a limited set of automated program transformations, and new threats are constantly appearing. Sometimes developers will wait for the automated program transformations to appear, other times they will remove the vulnerabilities manually, perhaps redoing that work with automated program transformations once they are available. Redoing the work may show some vulnerabilities that the programmers missed.

Application updates are typically distributed to end users with software patches. A patch can apply to source code or binary. We refer to a software patch that is produced by traditional security engineering approaches as a security patch, and a software patch that is the result of applying a program transformation as a program transformation patch. A program transformation patch is automatically created, while a manual security patch is created by a developer. Sometimes they are the same. However, a tool may allow a developer to focus on removing a vulnerability from an entire application; otherwise a developer will have attempted to manually remove a vulnerability from the few places where it has been reported. It is much easier for a developer to create a patch using a tool.

When program transformation tools are not readily available, application developers can respond to a vulnerability report with a quick fix: a security patch. At the same time, transformation tool vendors will start developing a program transformation tool. Once the tool is developed and thoroughly tested, application developers will apply the tool and distribute a program transformation patch that globally cures the vulnerability.
4.4 Advantages of New Security Engineering

The modified security engineering methodology has the following advantages:

- **Feasible Change.** Changing software, both in the development and the maintenance phase, will be more feasible with automated tools.

- **Greater Coverage.** Unlike a security patch, a program transformation patch will not be limited in scope; it will remove all instances of a security vulnerability.

- **Careful Development.** Unlike patch writers, developers of program transformation tools will not work under time pressure. The development deadline of a program transformation tool will not depend on the deadline to build a security fix. The program transformation patches can be tested thoroughly for completeness and the absence of regression errors.

- **Easy on the Users.** The number of transformation patches will be much less compared to the number of security patches. A vulnerability may appear at multiple places of source code; a program transformation patch will fix all instances of it, thus reducing the number of patches. It will be easy for end users to track fewer patches and apply them.

4.5 Applying Program Transformations

In order to explain how security-oriented program transformations can change traditional security engineering approaches, we return to the mail transfer agent case study (chapter 2). The case study narrates the difficulties of manually retrofitting security. Suppose sendmail developers want to secure it from its 3 main vulnerabilities. First, an attacker can compromise sendmail and get control of the whole system, because sendmail runs a monolithic program with root privilege\(^3\). Second, an attacker can use a compromised process to corrupt files by appending garbage data. Third, an attacker can launch a buffer overflow attack, because sendmail uses unsafe string functions. To solve these problems, developers would want to partition sendmail into multiple processes, most of them running with limited privilege (similar to qmail and Postfix). They would want to run the critical processes in a constrained environment (similar to Postfix). Finally,

\(^3\)It is possible to configure recent versions of sendmail to perform non-critical tasks as a less privileged user. But the complex configuration language influences many bad configurations where the sendmail program never drops privilege.
they would want to check for buffer bounds to eliminate buffer overflow vulnerability (similar to qmail and
Postfix).

Suppose sendmail developers had been able to use automated security-oriented program transformation
tools. When they first started getting buffer overflow vulnerabilities, they may have decided that the problem
was not just a particular line of code, but the fact that the standard string library is so easy to misuse. They
can have replaced the library by applying a library replacement transformation, perhaps developed by tool
vendors who write IDEs for C/C++. However, if sendmail developers are like most programmers, they
would not have done this the first time they found a buffer overflow vulnerability. But buffer overflows were
reported to them five times! Surely after two or three times, they would have decided to use an automated
transformation if one had been available, eliminating most of the security violations.

The purpose of many sendmail patches was to improve security checks. sendmail ran as superuser
so that it can deliver mail to any user. Before any read or write, it would check to make sure that it had
permission to access the file, since it can not count on the operating system enforcing security. It was hard
to make these checks perfectly.

None of our program transformations ensure the correctness of checks, though some of them help cen-
tralize checks so that they are more consistent and easier to change. However, the real problem with sendmail
was that it ran as superuser. It should be partitioned into a small part that needs to run as superuser and a
much larger part that does not. Both qmail and Postfix divide the mail delivery task into two parts. Most
of the system does not need to run as superuser. When mail needs to be delivered locally, it is given to a
“local mail delivery” function, which is a superuser. It determines the UID of the local user, changes UID to
become that user (and so no longer is superuser) and delivers the mail. sendmail could have been changed
to do this by using a partitioning transformation.

Although partitioning makes the system safer, it is still possible for the program to improperly read and
write public files. The solution that Postfix uses is to run the system inside a chroot jail. sendmail could
have been changed to do this with a transformation that runs a process inside a chroot jail\(^4\). Partitioning and
jailing sendmail would also eliminate many of the buffer overflow problems that do not originate from the
use of unsafe functions (hence unsolved by a library replacement transformation). These buffer overflow
problems would be bugs, but probably would not be system vulnerabilities, because the system running at a

\(^4\)This is the chroot Jail transformation in our catalog.
lower privilege level would not allow attackers to exploit them.

Even with automated program transformations, the sendmail developers would still have had to issue patches as their system evolved. If the transformations that they had used had all been standard Unix ones, they may have issued updates as an application of one or more transformations, rather than as a regular patch. This may make it easier for people making extensions to sendmail to move to the latest version, but it wouldn’t have made much difference to the average users, who doesn’t make extensions.

4.6 Issues of Applying Program Transformations

Security-oriented program transformations raise issues that affect how transformations are automated, how they are applied and how they are composed. We will refer to the relevant issues when we narrate individual security-oriented program transformations.

- **Granularity.** Different security solutions apply at different granularities of a system. A program transformation applicable at the architectural level will be harder to automate than a transformation applicable at the program level.

- **Specific Context.** Each program transformation has a specific context; it is applicable to a specific program representation. Different programming languages and platforms will have different sets of program transformations. Conversely, the mechanism of a program transformation can be different for different programming languages and platforms. Some program transformations apply to source code, some apply to binaries, while some apply to both.

- **Detail of Specification.** Program transformations are semi-automatic; users will have to specify the behavior of the target program. Different program transformations require different amount of specification. An architectural transformation will require more specification than a transformation that affects each part of the program.

- **Defense in Depth.** Applying one security solution does not guarantee security. A system should follow the principle of defense in depth [210]; it should have multiple layers of security tactics, instead of a single security strategy. Composing different security solutions is hard. Some program transformations will be orthogonal, while others may be useful only if applied in a specific way.
• **Composite Transformation.** Some program transformations are composite of many primitive transformations. They are similar to a large refactoring, which is composed of many small refactorings. If a developer can apply a large transformation instead of interactively applying small steps, it will be easier. He/She will also not have to worry about the order of transformations. However, creating a tool for a large, composite transformation is difficult. Hence, most of the large transformations are performed in small, interactive steps; these steps are then automated.

• **Behavior Preservation.** Program transformations to introduce security are not behavior preserving, according to its traditional definition. But the concept of behavior preservation should not be limited to external behavior preservation, it should mean that a program preserves the correct behavior. Incorrect behavior will change. A transformed program retains the same behavior for a user who is using it according to its specification, but it changes the behavior for a malicious user to prevent him from exploiting a security vulnerability. Program transformations to introduce security are behavior preserving (or perhaps good path behavior preserving) according to the more liberal definition.

• **Regression Test.** Security-oriented program transformations introduce new behavior for aberrant cases, the normal behavior should not change. Hence original test cases should run as before. However, there will be some corner cases in which the behavior will change. The developers will have to write new tests to eliminate regression errors hidden in the changed behavior.

### 4.7 Summary

This chapter introduced security-oriented program transformations, identified potential advantages of using program transformations, and narrated issues that affect how security-oriented program transformations are applied. The next chapter describes how the catalog of security-oriented program transformations has been derived from our previous work on security patterns.
Chapter 5

Creating a Catalog of Security-oriented Program Transformations

We have created a catalog of security-oriented program transformations that currently contains 37 program transformations. The program transformations in the catalog can be applied to fix the most important security problems faced by today’s developers. The transformations are grouped together by various classification schemes. The way the transformations are organized tells a user which transformation(s) to use to solve a security problem, and how to compose program transformations for defense in depth. This chapter describes how the program transformations in the catalog were derived, and how they are organized.

5.1 Deriving the Catalog

We derived our security-oriented program transformation catalog by surveying the problem domain and solution domain of security. We identified the most important security problems by surveying various vulnerability-trend reports. Then we identified the solutions to solve these problems by surveying the security patterns literature. This was aided by the fact that we had been maintaining a comprehensive catalog of all published security patterns. The pattern catalog contributes to most of the program transformations, but not all. The remaining program transformations describe how to automate well-known security solutions.

Our approach lies in the middle ground between two extreme approaches of deriving a catalog. One extreme is building a catalog by arbitrarily listing a number of security solutions. While it is easy to adopt, the resulting catalog does not provide any confidence in terms of coverage: it may miss many important opportunities of transforming a program. The closest analog of our program transformation catalog, the refactoring catalog, was built this way; the catalog was later extended by other practitioners with new refactorings. On the other hand, the most rigorous way to build a catalog of program transformations is to enumerate every possible security problem and their solutions, build tools to implement candidate solutions as program transformations, allow developers to use the tools and identify which transformations are more
useful, and finally include them in the catalog. It is impossible for a researcher to follow this rigorous path; building tools and user testing each transformation in a catalog would take hundreds of person years.

Our approach analyzes the problem domain and solution domain. However, instead of finding all possible security problems, we have restricted ourselves to the most important security vulnerabilities, identified by the vulnerability trend reports. On the other hand, we have searched for security solutions in our security pattern catalog, that manifests on the collective experience of the security patterns community. The pattern catalog is a summation of solutions described by many researchers; it covers the solution space better than any arbitrary approach.

Security-oriented program transformations are vulnerability-specific, but there is no one-to-one relationship. As such, there are multiple program transformations for a security problem\(^1\), and a single program transformation can describe solutions for multiple security problems\(^2\). Another approach for deriving the program transformation catalog can explore the general strategies of transforming program components and connectors, and list them as the program transformations. These building blocks can then be composed to create vulnerability-specific program transformations. This follows Spitznagel et.al.’s [193] approach of defining connector transformations as a composition of 8 basic transformations. The domain-independent building blocks are a principled, compositional means of systematically constructing connectors.

There are two reasons why we have chosen a vulnerability-driven approach instead of Spitznagel’s [193] approach of finding domain-independent building blocks. First, our goal is to explore security-oriented program transformations for the most important security problems. Naturally, this leads to exploring the problem domain, i.e., security vulnerabilities. To minimize the size of the catalog, we have explored vulnerability-trend reports to minimize the size of the problem domain. We have also explored security patterns, that describe general-purpose solutions for security problems. We eliminated similar or overlapping solutions when we developed our pattern catalog (see section 5.1.2.2). Thus, there has been a conscious effort throughout the process to minimize the number of program transformations and remove overlaps.

Second, our program transformations are distinct to the point that even an exploration of building blocks will create this many transformations. There are several program transformations in our catalog that add a new component, e.g., an authentication enforcer component, an authorization enforcer component, a peri-

---

\(^1\)There are several program transformations that fix various aspects of buffer overflow vulnerability. Section 12.1 describes them.

\(^2\)Variants of a Safe Library Replacement transformation (see section 7.4.3) target different vulnerabilities, e.g., buffer overflow, integer overflow, race condition attack, SQL injection, etc. All of them are listed under one transformation.
ter filter component, etc. Each of these transformations is distinct, because it creates a component by composing classes of a specific security framework, or it composes a component from a specific template. Cataloging them as a ‘compose framework class’ transformation, for instance, does not guarantee general-purpose applicability similar to the connector transformations, because the transformations are dependent on the specific security problems.

Some program transformations share common steps, but their end goals are vastly different. A **Partitioning** transformation distributes components into various components and modifies inter-component communication mechanism. Many of the same steps are applied in a **Single Access Point** transformation, that distributes components to extract access points of an application. The apparent similarity in their mechanisms is complicated by the goals of these two distributions, which means the analysis and the implementation mechanism will be different. Furthermore, there are other steps in these transformations that are different. Identifying the building blocks of all the transformations will create an even larger catalog.

Besides, describing program transformations in terms of vulnerabilities makes it easy for application developers to apply the transformations. A tool builder’s perspective may prefer that the catalog contains building blocks that are composed into security solutions. Here, we have taken an application developer’s perspective. Our aim is to understand the feasibility of introducing security solutions as program transformations; a domain specific approach that explores security vulnerabilities is more appropriate for this study. The tool builders perspective is a logical next step of implementing program transformations in our catalog.

Not all the security-oriented program transformations derive directly from security patterns. Security solutions such as encryption, signature, etc have been used in the solutions of many security patterns, but there are no specific security patterns that describe these\(^3\). These security solutions have also been described as security-oriented program transformations.

Our security pattern catalog has 91 entries. These patterns contribute to 30 program transformations. The remaining 7 program transformations originate from well-known security solutions.

\(^3\)There is one security pattern paper [26] that describes patterns for cryptography. But the patterns in the paper were not documented well. Hence they were not included in the security pattern catalog.
We will describe how we have chosen the security-oriented program transformations, both from the pattern catalog and from existing security solutions. The entire activity can be summarized in four steps:

1. Analyzing the problem domain and identifying relevant security problems.
2. Analyzing security pattern catalog for candidates that can be described as program transformations.
3. Identifying other well-known security solutions that can be described as program transformations.
4. Describing the mechanism of security-oriented program transformations.

The following sections describe these steps.

5.1.1 Analyzing the Problem Domain

We have studied various vulnerability trend reports and analyzed the threat model of various software applications to find the important security problems. We have identified 5 important vulnerability classes: input validation vulnerabilities, authentication vulnerabilities, access control vulnerabilities, error handling vulnerabilities, and denial of service vulnerabilities.

Our vulnerability classes have been derived primarily from the OWASP list of top ten security vulnerabilities, and have been supported by other vulnerability trend reports. OWASP [151] identifies top ten security vulnerabilities that affect modern systems. They are, unvalidated input (A1), broken access control (A2), broken authentication and session management (A3), cross site scripting flaws (A4), buffer overflows (A5), injection flaws (A6), improper error handling (A7), insecure storage (A8), denial of service (A9), and insecure configuration management (A10). Among these, cross site scripting (A4), buffer overflows (A5) and injections flaws (A6) fall under the umbrella of unvalidated input (A1). Securing storage (A8) has two concerns: deciding the appropriate cryptographic solution and implementing proper access control with the cryptographic solution. Deciding the appropriate cryptographic solution is entirely a manual activity that is part of requirements engineering, while implementing proper access control is the same as fixing access control vulnerability (A2). Securing configuration management (A10) is related to software maintenance rather than design. Thus we have arrived at our 5 vulnerability classes.

Other vulnerability trend reports support the OWASP list. The CERT list of top twenty most severe vulnerabilities of all time [207], and SANS list of the top twenty security risks in 2007 [180] follow the
The OWASP list. MITRE [45] have identified 41 different types of security problems. Of them, 16 fall under the umbrella of injection attacks, 7 are related to access control and perimeter security, 6 include various types of denial of service (DoS) attacks, and 6 include attacks that originate from bad architecture or design. The remaining are general attack types that are being retired.

All of these vulnerabilities have been related to the Internet. Two factors contribute to this. First, the Internet subjects software systems to unforeseen security requirements. Even systems carefully designed for security face new security threats that their designers never thought about. Second, the Internet facilitates zero day vulnerabilities\(^4\).

It is possible to secure an application by keeping it isolated from any network, but such applications are almost useless. Symantec threat report for 2008 [200] documents 5491 reported vulnerabilities. 63% of the vulnerabilities affect Internet based applications, the remaining applications are also accessible by some network. For 74% of all vulnerabilities, the exploit codes are widely available.

We have used the OWASP classification of security vulnerabilities as the starting point. The next step is to identify security solutions that fix the security problems.

### 5.1.2 Finding Program Transformations from the Security Pattern Catalog

A security pattern describes a solution that experts use to solve a security problem in a context. We have been maintaining a comprehensive catalog of all published security patterns [92]. The security pattern catalog currently contains 91 patterns. The catalog is a union of all security patterns that appear in many books, catalogs and papers on security patterns. It accumulates the experience of the entire security pattern community and is a fair representative of the solution domain of security. 30 out of 37 transformations in our catalog are derived from security patterns.

This section describes our catalog of 91 security patterns, organization of the catalog, and how we have selected 30 program transformation candidates from the pattern catalog. The security pattern catalog has been derived by surveying the available literature on security patterns and removing security patterns that are described in multiple catalogs. We will describe how the patterns are organized in the catalog. We will also describe how the organization helps in finding the candidate security patterns that can be described as program transformations.

\(^4\)A zero day vulnerability occurs when a flaw in a software code has been discovered and exploits of the flaw appear before a fix or patch is available. The exploit is quickly available to the script kiddies.
5.1.2.1 Security Pattern Sources

The security pattern catalog is derived by surveying the available literature, which includes books, pattern catalogs and pattern papers.

There have been 3 books on security patterns published between 2005 and 2006. Markus Schumacher led a working group on security patterns to write a security pattern book [184] in December, 2005. This book had 46 security patterns from the domain of enterprise security and risk management, identification and authentication, access control, accounting, firewall architecture, and secure internet application. This book aggregates many previous works on security patterns, including the first catalog of security patterns [220], Markus Schumacher [185] and Eduardo Fernandez’s [27, 68] work on security patterns etc.


Microsoft’s Patterns and Practices group published a security patterns book [101] with their release of Web Services Enhancement (WSE) 3.0. The book is a guide for using WSE 3.0 in Web services development. It lists 18 patterns: 5 for authentication, 2 for message protection, 5 patterns for transport and message layer security, 2 for resource access, 3 patterns for service boundary protection and 1 for service deployment. These patterns are all described from Microsoft technology perspective.

People have worked on pattern catalogs covering various domains. In 2002, Bob Blakley and Craig Heath compiled the first draft of their catalog of security patterns with the members of the Open Group forum. In 2004, they released a revised version of the catalog [21] containing 13 patterns: 5 patterns for improving reliability, and 8 for security.

Kienzle et al. [118] listed 26 patterns and 3 mini patterns in their security patterns catalog. The patterns were described under two broad categories. There were 16 structural patterns (including the 3 mini-patterns) and 13 procedural patterns.

In 2001, Sasha Romanosky [173] compiled a collection of 8 design level security patterns. These patterns cover the security threats in different application domains, e.g. enterprise management (Risk Assessment and Management), brokered communication (3rd Party Communication), testing (White Hats, Hack Thyself), fault tolerance (Fail Securely), software maintenance (Low Hanging Fruit) etc. Sasha Romanosky [174] presented another collection of enterprise level security patterns in 2002.
We have created two pattern catalogs. The first stems from our study of the evolution of mail transfer agent architecture [94, 83, 84, 85, 93]. This contains 11 security patterns, and 4 reliability patterns. These patterns are not specific to mail transfer agents; they can be used by the designers of other secure systems. Our second catalog compiles a collection of 8 privacy design patterns [86].

### 5.1.2.2 Removing Overlaps

There is a significant amount of overlap in the different security pattern catalogs. For example, patterns for authentication have been listed in a number of pattern catalogs under different names.

- The Authenticator pattern, listed in conjunction with other patterns of operating systems security [68], was later included in the Wiley security patterns book [184]. Authenticator is a type of Policy Enforcement Point [220], that intercepts the interactions of a subject with the system and applies a protocol to verify the identity of the subject. The protocol that is used might depend on some user input (i.e. something that the user knows e.g. password, or possesses e.g. a smart card, or has e.g. a biometric characteristic), or the input provided by a trusted third party (i.e. brokered authentication).

- Sasha Romanosky described the same pattern under the name Security Provider [173].

- Kienzle et al. lists it as Authenticated Session [118] in their security patterns.

- The Core Security Patterns book described the Authentication Enforcer pattern [194] from Java tool-oriented perspective. This pattern creates a centralized authentication enforcement that performs authentication of users and encapsulates the details of the authentication mechanism.

- Microsoft security patterns book [101] also lists this pattern. The authors described authentication in terms of various strategies supported by the WSE 3.0 framework. They divided authentication in two parts and described two patterns. The Direct Authentication pattern describes the scenario when web services act as the authentication service to validate the subject credentials. The Brokered Authentication pattern, alternatively, provides authentication without the direct communication between the subject and the service. Three different brokered authentication strategies were described as separate patterns, namely, Brokered Authentication: Kerberos, Brokered Authentication: X509 PKI and Brokered Authentication: Security Token Service (STS).
In the previous section, we have listed various sources of security patterns. Together, these sources describe 174 security patterns, but with many overlaps. Our security pattern catalog removes these overlaps. Currently, the catalog has 91 patterns [87, 88, 90].

5.1.2.3 Organizing Security Patterns

We have organized the security patterns [92] so that practitioners can identify the most appropriate pattern to solve a particular security problem. However, this organization also helped us identify the security patterns that are suitable candidates for program transformation. In this section, we will describe our pattern organization scheme. We will also list other schemes of organizing security patterns. In the next section, we will describe how the program transformation catalog was derived.

There are two aspects of organization [92]: classification and navigation. Classification means grouping patterns into small, correlated sets. However, good organization also facilitates navigation; it cross-references the related patterns and guides a user among them so that the user knows how to use a pattern. We have classified our pattern catalog using a combination of Zachman framework and threat modeling. We have correlated our patterns by describing them as a pattern language.

Many researchers have used the Zachman framework for classifying security patterns [184, 105, 100, 204]. Zachman framework [221] was introduced in 1987 as a table. The 5 rows of the table describe the levels of information model from the perspective of various stakeholders, e.g. the customer or the owner, the designer, the builder etc. The 6 columns describe different ways of describing an artifact: data (what?), function (how?), network (where?), people (who?), time (when?) and motivation (why?).

We have used a classification scheme based on the Zachman framework. Our tabular classification scheme uses the ‘Enterprise Architectural Space Organizing Table’ [204]. Out of the 7 columns of the table, 6 are from the Zachman framework table. The seventh column originates from the principles of test-driven development [15]. This column includes the patterns that validate the artifacts contained in the function column of the table. The rows describe stakeholders similar to the Zachman framework, but they have been specified in finer granularity using the architectural standards description from IEEE 1471 [107] and ‘Enterprise Architecture Framework’ [79].

Table 5.1 shows how the security patterns are classified using this tabular scheme. As an example of classification, consider the Safe Data Structure [94] pattern. This pattern is applied to remove the array
<table>
<thead>
<tr>
<th>Perspective [Table Rows]</th>
<th>Stakeholder [Table Rows]</th>
<th>Viewpoint [Table Columns]</th>
<th>Pattern count</th>
<th>Example Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration Architecture</td>
<td>Enterprise Architect</td>
<td>Function</td>
<td>2</td>
<td><em>Single Sign On.</em> [118] Allow a user to access multiple services in a distributed network environment without having to re-authenticate on every request.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Function</td>
<td>22</td>
<td><em>Single Access Point.</em> [220] Single Entry Point for each process.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Network</td>
<td>1</td>
<td><em>Secure Communication.</em> [21] Create secure channel.</td>
</tr>
<tr>
<td></td>
<td>Designer</td>
<td>Network</td>
<td>7</td>
<td><em>Stateful Firewall.</em> [184] Filter traffic based on state information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data</td>
<td>13</td>
<td><em>Encrypted Storage.</em> [118] Server data is protected by encryption.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Function</td>
<td>28</td>
<td><em>Server Sandbox.</em> [118] Servers run with least privilege to limit client activities.</td>
</tr>
<tr>
<td></td>
<td>Developer</td>
<td>Data</td>
<td>1</td>
<td><em>Safe Data Structure.</em> [94] Memory buffers contain length information that is checked before allocation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test</td>
<td>1</td>
<td><em>White Hats Hack Thyself.</em> [173] Test the system’s security by attacking it.</td>
</tr>
<tr>
<td></td>
<td>System Architect</td>
<td>Function</td>
<td>1</td>
<td><em>Low Hanging Fruit.</em> [173] Get quick fixes rather than trying to re-design the system every time a vulnerability is found.</td>
</tr>
</tbody>
</table>

Table 5.1: Classification of security patterns using the Enterprise Architectural Space Organizing Table

bounds checking vulnerability in a programming language with no garbage collection. A system written in C is vulnerable to buffer overflow attacks because of unsafe array operations, e.g. unsafe string handling. The *Safe Data Structure* pattern advocates the inclusion of length and allocated memory information with a data structure. This pattern is considered in the development phase of an application when the safe string processing libraries are written or re-used. Hence this pattern fits into the cell defined by the *Developer row* of *Application Architecture* perspective and the *function* column.

We used a hierarchical scheme based on threat models in combination with the tabular scheme for classifying security pattern. The reason for introducing a hierarchical scheme is that the flat, tabular space fails to distinguish between the scope of patterns. The tabular scheme could not classify 8 patterns because they fall into multiple cells in the table. For example, the *Defense in Depth* [210] pattern advocates the use of security checks in multiple layers of the application. Defense in depth is more of a security principle and therefore its context is defined by many rows and columns of the table. A hierarchical classification scheme can classify and distinguish these high level patterns from low level patterns. Figure 5.1 shows our hierarchical classification scheme and how it classifies the security patterns listed in table 5.1.
Our tree-based scheme allows us to classify a pattern by placing it on a leaf node (low level patterns) as well as internal nodes (high level patterns), therefore creating the hierarchy. It uses the STRIDE threat model [199] in all the branches.

We have correlated the security patterns by using a pattern language [3]. No pattern is an island. Patterns form a chain in which each pattern is derived from more general patterns and depends on the more specific patterns it contains. One pattern creates a problem that another one solves. Our pattern language [92, 87, 88, 90] correlates the patterns and instructs a user on how to use the patterns together to have defense in depth. It also describes the order in which certain patterns have to be applied to solve a problem.

Our pattern organization scheme is more systematic compared to previous methods that targets pattern classification only. The Open Group security catalog [21] classifies the patterns into two broad groups based on applicability: patterns for protected systems and patterns for available systems. Kienzle et al. [118] describes their patterns under two broad classes: structural patterns and procedural patterns. Both of these schemes are too broad to be useful. Steel et al. [194] classify their patterns according to logical architecture tiers, namely web tier, business tier, web service tier, and identity tier. This classification scheme has the advantage that the partitioning is aligned with the system tiers. Hence the classification does not introduce new vocabulary for system architects and developers. However, many security patterns describe solutions.
that are not limited in one system tier. Finally, people have used a modified version of the Zachman framework by adding a column representing the security view [105, 100]. The security view addresses all model levels, from the enterprise scope to the detailed representations. But putting all security patterns in one column creates a one-dimensional scheme, only classifying based on the stakeholders.

Our organization scheme combines classification with navigation. Our intent was to help users in understanding the pattern space better. However, when we explored the security pattern catalog to find candidates for security-oriented program transformations, our organization scheme helped us identify the suitable candidates and allowed us to describe the relationship between program transformations. The next section describes how the program transformation catalog was derived from the security pattern catalog.

5.1.2.4 Security Patterns to Program Transformations

Our pattern catalog has 91 entries. We trimmed our security pattern catalog and found 30 security solutions that could be described as program transformations. This was done in two steps.

1. We used the tabular classification scheme for security patterns to identify patterns that describe solutions applicable to software. This step resulted in 74 patterns.

2. Next, we analyzed the implementation mechanism of each of the remaining patterns. Some of the patterns describe high-level policies, some describe solutions that are too difficult to implement; also there are patterns that describe the same technique applied in different contexts. After removing these, we had 29 patterns that contributed to 30 security-oriented program transformations.

We will describe these two steps in detail.

Using Pattern Classification Scheme to Find Relevant Patterns. Not all security patterns are about code and software design. The tabular classification scheme for organizing security patterns is used to identify patterns that are relevant to software. Table 5.1 shows how security patterns are classified using enterprise architectural space organization table. From the table, patterns that are in rows grouped by application architecture perspective, and are in the function and data column are suitable candidates for program transformations. This constitutes 74 patterns listed in figure 5.2.

Among the remaining 17 patterns, 7 are about software processes and 10 are about hardware. There are security patterns for asset evaluation, risk assessment, and threat modeling [184] that describe a process,
and could not be described as a program transformations of software artifacts. Patterns such as *Third Party Communication* [174] describe the process of creating and retaining trust between two companies in a business relationship. *White Hats, Hack Thyself* [174] security pattern describes the process of engaging gray hat hackers into testing the security of a software application.

Another group of patterns that are not related to code and software design are patterns that are related to hardware. *Packet Filter Firewall* [184] and *Stateful Firewall* [184] patterns guide practitioners how to select the most appropriate firewall, while *Demilitarized Zone* [184] pattern guides how to deploy multiple firewalls.
Finding Patterns that could be Described as Program Transformations. Among the remaining 74 patterns, 13 describe very high level solutions, 22 describe solutions that are hard to implement, and 9 describe solutions are common to other patterns. Removing these leaves 29 security patterns, that contribute to 30 program transformations. Figure 5.3 shows how the 74 patterns are partitioned.

13 patterns prescribe high level solutions. An example of such a pattern is Defense in Depth [210]; it is a security principle that is fundamental in any security design, but it is too general to form a program transformation. Another example is the Anonymity Set [86] pattern. It is a core pattern for privacy pre-
serving applications. It suggests that the fundamental principle of keeping information private is to mix it with other information and make private information indistinguishable. But implementing the high level solution depends on the context of an application; it cannot be expressed as a general purpose program transformation.

22 security patterns provide solutions that are too complex. Limited Access [184] and Full Access with Errors [184] are two patterns that describe how secure user interfaces can be designed. For an application with many users with different privilege levels, one option is to provide each user limited view to the user interface. A user only sees the options that his/her privilege allows. The other option is to show all functions to a user, but only allow him/her to access the functions that are permitted. Graphical user interface is difficult to code and almost impossible to code automatically. Therefore, automatically adapting an existing user interface to follow either options is not possible. Another example is the Role Based Access Control [184] pattern. It is a complex pattern that introduces massive, context sensitive changes in the code. It is hard to describe a general solution in terms of program transformations.

In 10 cases, the solution mechanism of multiple security patterns are the same; thus multiple security patterns correspond to one program transformation. For example, input filtering is an important protection mechanism that is motivated by Content Dependent Processing [93] pattern. The filtering mechanism in an application follows the Pipes and Filters [35] architectural pattern: an input variable passes through a series of filters. The same mechanism applies to other cases when data has to be modified, e.g., encoding or decoding data, encrypting or decrypting, using parity for error detection and correction, or padding data packets so that all packets are of the same length. Each of these cases is defined by its own security pattern, but the corresponding security-oriented program transformation is the same. Similarly, Partitioning program transformation in our catalog is a combination of Compartmentalization [210] and Distributed Responsibility [208] patterns.

29 security patterns contribute to 30 security-oriented program transformations. Safe Data Structure [93] pattern contributes to 2 program transformations: Safe Library Replacement transformation and Safe Type Replacement transformation (see section 5.1.4 for details). Figure 5.4 lists the patterns and corresponding program transformations.
5.1.3 Security-oriented Program Transformations from Other Sources

Figure 5.4 lists 7 program transformations in our catalog that do not derive from security patterns. Instead they are security programming techniques that could be described as program transformations. Program transformations to create a message digest, or a message signature describe how classes of Java or .NET security API could be composed to perform the tasks. Similarly Encryption/Decryption transformation composes Java or .NET security API. There are security patterns that use cryptographic solutions (Client Data Storage [118] and Encrypted Storage [118] patterns shows in figure 5.3), but there are no specific patterns describing these cryptographic operations. Explicit Type Enforcement transformation adds missing type information to prevent integer overflow errors that originate from the use of integer datatype in an unsafe context. Guarded Object transforms a program to encapsulate each resource and the corresponding set of permissions in one object. Least Privilege transformation automatically analyzes a program to identify the lowest privilege level required. Fuzzing transformation adds user-directed white box fuzzing components.

The remaining part of this dissertation describes these program transformations in detail. Also a short summary of the program transformations is in appendix A.
5.1.4 Describing Mechanism of Program Transformations

The final step is defining the mechanics of the transformations. The mapping from a security pattern to a program transformation is not trivial. A security pattern provides a high-level description of a security solution. It does not describe how to implement the solution, nor the steps to transform a program to introduce the solution. For each candidate security pattern, we had to identify the implementation method and then describe the steps of program transformation. For example, a Safe Data Buffer [93] security pattern provides a solution to prevent buffer overflow attacks. Buffer overflow occurs because an unsafe language such as C does not check buffer bounds while performing a buffer operation. The Safe Data Buffer pattern says that one should check for length information before performing any operation on the data. It does not give any hint of how to implement the solution. The lessons from this pattern have been implemented by a number of safe string libraries (e.g. strlcat and strlcpy [141], libmib library [65], ISO/IEC 24731 [109], libsafe library [13] etc) and safe data types [147, 93]. We have described the solution as a Safe Library Replacement transformation as well as a Safe Type Replacement transformation. The Safe Library Replacement transformation finds all instances of an unsafe library in a program and replaces each instance with a corresponding safe library\(^5\). The Safe Type Replacement transformation replaces char * pointers with a safe data type that keeps length information of the buffer\(^6\).

In some cases, the pattern describes a high level solution, yet the program transformation implements a concrete solution. For example, Randomization program transformation randomizes a data variable in a program or instruction set of a system to make it distinct. This, as well as some variants of Safe Library Replacement, has been motivated from the Minefield [118] pattern. The pattern suggests that systems should be customized so that each instance is different; therefore, some variants will be resistant to an attack vector that compromises other variants of the same system. There is no general way to implement the pattern. The program transformations introduce concrete solutions for varying a system to make it secure.

Not all parts of program transformations can be automated. But if a program transformation can be automated, tools will make it easier for developers to apply them. Some program transformations can be completely automated, but most transformations are interactive and require some manual intervention.

\(^5\)Safe Library Replacement transformation is not limited to preventing buffer overflow attacks. It describes how to replace any unsafe library function with a safe alternative.

\(^6\)Safe Type Replacement is not limited to replacing char * pointers. It also replaces other unsafe data types.
5.2 Organizing the Program Transformation Catalog

Security-oriented program transformations vary in their impact on an existing program—some transformations make changes at a few parts of a program, others affect many parts of a program, while there are program transformations that go beyond the boundaries of a single system. These categories can be used as a scheme to organize the transformations. Another scheme to organize the transformations in the catalog is the type of change they make. Yet another scheme is based on the type of vulnerability that is fixed by a program transformation. Categorizing program transformations based on these common properties makes it easy to refer to families of related transformations, to learn the transformations in the catalog, and to find new transformations.

Our catalog of 37 program transformations is organized based on 3 orthogonal criteria: vulnerability fixed by transformation, impact of transformation, and type of transformation. Combining these criteria gives a three-dimensional model for categorizing transformations.

Organization based on Vulnerability Fixed. A security-oriented program transformation attempts to fix a vulnerability in an application. But not all program transformations are about fixing a vulnerability. We have organized the program transformations in 6 groups; 5 groups originating from the important vulnerability classes, and one group for fundamental program transformations that do not fix any particular vulnerability.

Section 5.1.1 describes how 5 distinct vulnerability classes have been derived from the OWASP [151] report of top ten security vulnerabilities. They are,

1. Unvalidated Input
2. Broken Access Control
3. Broken Authentication
4. Improper Error Handling
5. Denial of Service

There are some program transformations that do not fix any particular vulnerability. For example, Partitioning transformation splits a process into multiple processes so that each process can run with different
privilege level. This can be applied to split a monolithic process that otherwise needs to run with root privilege. The split processes can run with lower privilege level except for a few that absolutely need root privilege. If a buffer overflow vulnerability exists in the monolithic process and it gets distributed in a partition running with lower privilege, the buffer overflow vulnerability becomes uninteresting for an attacker because he cannot do anything malicious to the system after exploiting the vulnerability. Partitioning does not remove the buffer overflow vulnerability. It makes the vulnerability harder to exploit.

Security-oriented program transformations that do not fix any vulnerability are in fact fundamental for system security. These program transformations should be applied before other transformations. We have created a separate category named ‘fundamental transformations’ for these transformations.

The next six chapters of this dissertation describe six groups: fundamental transformations followed by five groups that solve five categories of vulnerability.

**Organization based on Impact of Transformation.** Program transformations in the catalog can also be organized using the impact of applying a transformation on a codebase. There are 4 types.

1. *Small, infrequent change.* Some transformations have a small footprint. The code change for each instance of a transformation is small; also the frequency of these changes in the total codebase is small.

2. *Small, frequent change.* The code changes for an instance of a transformation might be small, but the changes occur at many places of the code.

3. *Large change.* These transformations make large changes on a codebase.

4. *Change beyond system boundary.* Some transformations require code change beyond system boundaries, i.e. in multiple network elements.

**Organization based on Type of Change.** Another criteria for grouping transformations is based on the type of change they introduce to the system. We have identified 4 types of code changes. They are:

1. *Add Component.* Instantiate a component that adds a new security functionality and plug it in the existing system.

2. *Limit.* Set a limit on the resources and check it before a resource is used.
3. *Mutate data.* Change encoding of data, validate data, or rectify data to remove data related security vulnerabilities.

4. *Distribute.* The opposite of centralize is distribute. Break up a single entity into smaller parts, e.g. distribute a task between smaller subtasks, or break up a process into multiple processes.

### 5.3 Summary

This chapter describes how the program transformation catalog is derived and how it is organized.

37 program transformations in the catalog are organized into 6 groups based on the type of vulnerability they fix. The next six chapters describe these groups. The first group, fundamental transformations, includes program transformations that do not fix any particular vulnerability but are however core to securing an application. Program transformations in the unvalidated input group remove various type of data injection attacks. The third and fourth group consist of program transformations that fix access control and authentication vulnerabilities. The fifth group has program transformations that fix vulnerabilities originating from improper error handling. The final group contains program transformations that fix denial of service vulnerabilities. In each of the next six chapters, program transformations described in that chapter are further organized based on their impact on codebase, and the type of change they make.
Chapter 6

Fundamental Security-oriented Program Transformations

The security-oriented program transformations described in this chapter do not remove any particular security vulnerability. They are called fundamental program transformations because—1) they modify the architecture of an application and make many security vulnerabilities difficult to exploit, and 2) they should be applied before applying other vulnerability-specific program transformations.

We will describe 4 security-oriented program transformations in this chapter. They are, Partitioning, Least Privilege, Single Access Point and Policy Enforcement Point. These program transformations modify applications to reach two goals. Partitioning and Least Privilege transformations align application components with the required privilege level so that each component runs with the least privilege. The other two transformations minimize the access points and create a policy enforcement point so that appropriate authentication, authorization and input validation methods can be added to the access points.

Typically, security-oriented program transformations are distinguished from refactorings since they are not behavior-preserving. They prevent security exploits, and as such change the program behavior that an attacker experiences\(^1\). Since, fundamental program transformations do not remove any vulnerability, the changes that they make are closer to refactorings. They make architectural changes, many of which can reuse existing refactorings.

Consider the Partitioning transformation. It changes the program behavior in an implicit manner. When an application goes through a program transformation that removes a buffer overflow or an SQL injection vulnerability, an attacker can explicitly experience the change in the behavior; since he/she will not be able to reproduce the vulnerability. On the other hand, partitioning a C program splits a monolithic process into multiple processes, each of them running with separate privilege level. The behavior of the program changes, because of the change in the privilege level. However, an attacker can still reproduce a buffer overflow or an SQL injection vulnerability, unless the code containing the vulnerability is distributed to a

\(^1\)They retain the good path program behavior.
partition that runs with a lower privilege. The distribution does not directly fix a vulnerability, but it prevents an attacker from exploiting it. The program behavior changes implicitly.

Another architectural transformation similar to Partitioning is the Single Access Point transformation. It minimizes the number of access points of an application, so that there are less points to secure. A Policy Enforcement Point transformation introduces a component where authentication, authorization and input validation components can be introduced using other program transformations. These two transformations have significant impact on securing the system perimeter, but they do not change any behavior. Tools for these three fundamental transformations can be made reusing many existing refactoring tools. We will describe these issues when we describe how tools for these program transformations can be implemented.

The next two sections describe these program transformations. To describe the transformations, we have augmented the format [71] for describing refactorings.

### 6.1 Privilege Separation

Every program component should run with the least privilege level [179], otherwise it is vulnerable to various attacks. Some operating systems and programming languages enforce various measures for preventing privilege escalation, but the most common approach is partitioning an application. By partitioning, an application is split into several compartments, each of them running with separate privilege level. This creates a sandbox for an attacker: after a successful attack, an attacker is limited in a partition since he does not have necessary privilege to affect other partitions.

The main goal of partitioning is to create privileged compartments and separate them from unprivileged compartments. An ideal partition factors security-critical tasks to a few partitions, so that other partitions can run as a less-privileged user. The critical parts run with a higher privilege, but they are smaller in size and are easier to understand and test thoroughly.

Separating privileges by partitioning application components is a fundamental security countermeasure; it does not remove any particular vulnerability. For example, early versions of sendmail run as a monolithic process with root privilege. When an attacker compromises a sendmail process exploiting a buffer overflow vulnerability, he/she can compromise the entire system because of the root privilege [93]. The first sendmail vulnerability [32] exploited by the Morris worm [63] was a buffer overflow vulnerability that allowed an attacker to get control of the system, because it was running as root. By partitioning, compartments can run
with the least privilege required. Thus an attacker can find a buffer overflow error, but it will not lead to an exploit because of the limited privilege.

The first transformation described in this section is Partitioning. The underlying mechanism behind partitioning is to split application components into separate compartments and then introduce remote communication among compartments. Partitioning is an architectural transformation. It requires a lot of specification, e.g. distribution of functionality, remote communication mechanism, privilege level for each partitions, etc. Finding the appropriate privilege level of a component is part of the Partitioning specification, but it is a program transformation itself. The Least Privilege transformation describes how program analysis techniques can be used to determine the minimum privilege level of a process. Least Privilege and Partitioning transformations go hand in hand.

6.1.1 Partitioning Transformation

You want to design a system that constrains a security failure in one part from affecting another part.

*Partition the system into multiple security domains.*

**Motivation**

A single program with too many tasks is insecure. Such a program runs with the privilege suitable to do the most security-critical task. To get around this, one can distribute the tasks of a program into multiple logical partitions so that processes in each partition run with a single privilege level. Processes in one partition are isolated from processes in another partition, if the privilege levels are different. Thus, a security failure in one part is constrained from affecting another part. Partitioning hardens the architecture that rules out many security vulnerabilities from happening.

From our mail transfer agent case study, sendmail runs as one big program with root privilege, although
it only needs the privilege for email forwarding. An attacker, after compromising sendmail, can control the whole system. Postfix, in contrast, distributes its tasks among multiple processes with limited privilege.

One cannot secure a program by arbitrarily partitioning it. If tasks are not cleanly distributed, a program does not become any more secure than it initially was. Worse, a program might become inefficient because of increased remote communication, or even less understandable. Additionally, partitions should follow the principle of least privilege [179].

**Preconditions**

Partitioning is an architectural transformation. A program benefits from partitioning when,

- The program composes multiple privilege levels. It runs with the highest privilege level, and drops its privilege while performing low privilege functions.
- The functions to be placed in different partitions share very little state information.
- Remote communication does not leak information.

**Mechanism**

A developer specifies functional distribution, partition interfaces, privilege levels and inter-process communication mechanism.

The program transformation partitions according to this specification. It distributes functions and privileges so that privileged functions can run in partitions with a higher privilege level, while others run with lower privilege. The partitions are chosen such that they minimize inter-process communication. Various inter-process communication mechanism can be chosen, e.g. CORBA, pipes, sockets, RMI, etc.

A partitioning transformation requires a lot of specification. Identifying the appropriate partitioning for a system and specifying the partitions is a manual activity. Once a specification is available, the transformation tool automatically creates the partitions.

**Example**

We describe two examples of partitioning here. First, we will show the impact of Partitioning on a small C program. Then we will describe how to partition a Java program by describing a case study.
Partitioning a C program. We illustrate the Partitioning transformation with a small C program that has many of the same design features as sendmail. After receiving mails, sendmail stores them in its queue and spawns another process for delivery. The example C program gets input, writes it to a file, and spawns another process to print the stored information to stdout. The conciseness of the program highlights the changes made by each transformation.

The C program reads from an input file (lines 34~37) and stores it in a new file inside a storage directory
The program uses the process id as the file name (38∼41). Then it spawns a child process and returns (45∼48). The child process reads all files in the storage directory (10∼17), prints file contents (18) and deletes the files (19∼20).

The C program runs as a monolithic process. If it runs as root, an attacker can compromise it and get control of the system.

In order to partition the example program, we specify:

- The main and the ChildProcess functions should be in different programs in different logical partitions.
- There should be a reader program containing the ChildProcess function and a writer program containing the main function.
- The parent and the child process in the example communicates using shared files. The reader and the writer programs should communicate similarly.

A partitioning transformation produces a reader and a writer program. The writer program gets input from a file and writes that in a new file inside a shared directory. The following lists the writer program.

```c
int main (int argc, char * argv[]) {
    FILE* fp;
    char filepath[100];
    char fbuf[100];
    char line[80];
    char pidstr[15];
    pid_t pid;

    fp = fopen("data.in", "rt");
    while(fgets(line, 80, fp) != NULL)
        strcat(fbuf, line); // Read input file
    fclose(fp);
    pid = getpid(); // Get process id
    sprintf(&pidstr[0], "%d", pid);
    strcpy(filepath, "temp/");
    strcat(filepath, pidstr); // Formulate file name
    fp = fopen(filepath, "w"); // Create file for writing
   fwrite(fbuf, sizeof(fbuf[0]), strlen(fbuf), fp); // Write to file
    fclose(fp);
    return 0;
}
```

The reader program polls the shared directory. When the writer program creates a file, the reader program prints the content to stdout and deletes the file.
void ChildProcess()
{
    DIR *dirptr;
    FILE *fp;
    struct dirent *direntry;
    char filepath[100], dirpath[100];
    char buf[1024];
    char line[80];

    strcpy (dirpath, "..\writer\temp");
    dirptr = opendir(dirpath); // Open directory
    while (direntry = readdir(dirptr))
    {
        strcpy (filepath, "..\writer\temp/");
        strcat (filepath, direntry->d_name);
        fp = fopen(filepath, "r"); // Open file for reading
        while(fgets(line, 80, fp) != NULL)
        {
            strcat (buf, line); // Read file content
            printf("%s\n", buf); // Print file content
            fclose(fp);
            remove(filepath); // Delete file
            return;
        }
    }
    closedir(dirptr); // Close directory
    return;
}

int main (int argc, char * argv[])
{
    for (;;)
        ChildProcess();
}

qmail and Postfix manage their mail queue in this way. A writer process writes incoming mails to mail queue, and a reader process fetches mail from the queue and dispatches for local or remote delivery.

Partitioning a Java program. We have partitioned jftp version 0.3, an FTP server written in Java. jftp runs everything in the same virtual machine. We want to partition the program in order to extract a gatekeeper component. The gatekeeper component will communicate with outside users, we want to run it in a separate virtual machine with lower privilege level, separate from the internal part of the application that processes the requests and require higher privilege. After the partitioning the components communicate using RMI. The entire process was done using Eclipse IDE. We manually performed the steps; one can easily visualize that an automated tool can be built to perform these tasks.

Partitioning a Java program requires two steps. First, a developer specifies how the classes and methods are distributed. In jftp, we first refactored to extract the perimeter methods, i.e. methods that contain functionality to interact with external users. We used refactorings available in Eclipse IDE, e.g. Extract Class and Extract Method. Then the communication mechanism between the extracted gatekeeper component and the internal components is changed from local method calls to RMI calls. We created the stubs and
skeletons using existing tools; we have introduced RMI classes in the code manually, but this step can be automated.

Since this transformation extracts the gatekeeper component, we have used this same example for the *Single Access Point* transformation (section 6.2.1). There, we have described more details.

**Towards a Partitioning Tool**

Partitioning is a large transformation that involves two key steps: identifying the distribution and introducing remote communication. A partitioning tool should be interactive and perform each of the steps according to the developer specification. A number of specification-driven partitioning tools are available, but usually their goal is to get better performance [201, 203], better reliability [98] or better energy consumption [214]. Some of these tools apply program analysis techniques to model the control flow of an application. These analysis and visualization tools can provide a partitioning guideline for developers. At the same time, many of the existing refactorings can be used to create the distributions.

The specification for a *Partitioning* transformation includes the functional distribution, privilege level of partitions, and remote communication mechanism. The specification does not have to be textual; it can be the way a developer combines the steps of a *Partitioning* transformation. A developer can use refactoring tools to distribute and create the compartments, and then choose a remoting mechanism. The entire distribution can be done using several small steps of applying refactoring tools. The developer, by performing the refactorings, effectively specifies the distribution. *Partitioning* is applied together with the *Least Privilege* transformation (discussed next). This modifies the privilege level of the partitions so that they run with just enough privilege.

The simplest partitioning is to extract the security-critical functions of a program and run them in a separate privileged partition. Two tools that follow this principle are Privman and Privtrans. Privman [119] is a library of privileged functions that runs the functions in a privileged server. Privtrans [28] transforms an input program into a monitor and a slave program based on annotations identifying critical functions and data. The monitor performs all critical tasks. These tools prevent privilege escalation [165] in C programs.
Some specification-driven partitioning tools apply to programs written in Java, e.g., Pangaea [192] and Doorastha [55]. These tools make local and remote method calls syntactically and semantically indistinguishable. Other partitioning tools apply to Java bytecode, e.g., Addistant [201] and J-Orchestra [203]. These transform bytecode to automate the mechanism for remote references, such as replacing method calls with remote method calls, direct object references with proxy references etc. A non-Java but related partitioning tool is Coign [104], which applies to a binary built from COM components. Coign profiles inter-component communication during runtime, selects a distribution with minimum communication overhead, and automatically modifies the binary to produce the desired distribution.

Another partitioning approach has a different goal than function distribution based partitioning. A data-driven program partitioning approach distributes computation for protecting data confidentiality and integrity [222]. However, the implementation mechanism is the same. Programmers annotate source code with confidentiality and integrity policies; programs are then partitioned automatically to run securely on heterogeneously trusted hosts.

In order to figure out which part of a program goes to which partition, a developer has to possess deep understanding of an application domain. On the other hand, when this specification is supplied to a partitioning tool, it does not need to understand the purpose of the program, and in fact the analysis that it uses are not horribly complicated. This separation is necessary to build a general purpose tool; most partitioning tools mentioned in this section operate this way.

6.1.2 Least Privilege Transformation

You have a program that contains many functions, each of them requiring different privileges. You want to ensure that it runs with the lowest privilege.

Run the program with the lowest privilege required to perform all tasks.

Motivation

The least privilege principle limits the damage that can result from a security compromise. A system where applications run with just enough privileges can thwart an exploit because an attacker who has initiated a fault cannot do much with the limited privilege. It allows an administrator to identify the critical parts of a system; he/she can then take extra security measures to protect those few isolated parts.
Least Privilege transformation is typically applied together with Partitioning transformation. The main motivation of Partitioning is to isolate the critical components of an application, components that have to run with root privilege. For other components, the privilege is lowered so that they run with minimum privilege.

Privilege of a partition can be determined by various factors: resource access privilege, data access privilege, execution privilege, etc. This program transformation determines privilege in terms of resource access. This means the files accessed by a program or the sockets opened by a program. This metric does not necessarily determines the lowest privilege. However, the intent of Least Privilege transformation is to determine a separate privilege level for a partition created by the Partitioning transformation. Hence, the intent of this transformation is to create a new privilege level in terms of resource access, so that we have partitioning of privilege instead of least privilege.

In the mail transfer agent case study, qmail is an excellent example of applying the least privilege principle. qmail processes run under 5 different users, each of which has the minimum privilege required.

Preconditions

A program with the following characteristics benefits from a partitioning transformation.

- The program has been partitioned so that critical components are isolated into a few compartments.
- The functions to be placed in non-critical partitions require very low privilege level.
- The program accesses a fixed set of resources (files, sockets etc).

Mechanism

This program transformation involves two steps: 1) determining the minimum privilege level, and 2) modifying program to run with the lowest privilege.

In order to determine the minimum privilege level for a program, identify the resources used by it. Two types of resources are considered: files (including static and dynamic libraries) touched by the program, and sockets opened by a program. Find the sockets opened by a program by searching the source code. Find the files touched by a program. For a C program running in Unix environment, Search for fopen(3) system calls in source code for a list of resource and configuration files. Trace binaries with strace(1) command.
strace lists system calls made by the process. First, create a chroot jail and trace a program running inside the jail. Every time the program fails to open a file, it will log an error message in strace output. Parse these error messages, copy the listed files, and run strace again. Resolving some dependencies may open up more dependencies; repeat the whole process until there are no errors. The program analysis steps apply to both source code and binary.

Modify the program to run with the lowest privilege. Create a new user (and group), give the user ownership of a program and make the user own the resources used by the program. Alternatively, specify access control policies for program containers (e.g., policies specified for a Java virtual machine).

This program transformation modifies the environment of a program; the actual modification in the source code is very minimal. For a C program running in a Unix environment, the only modification in the program is the insertion of setuid calls to run the program under a custom user that has been created in the previous steps [42].

Example

Consider the writer program created after the Partitioning transformation (see the example section of Partitioning). The writer program has read access on the data.in file and write access on the temp/ directory. The program transformation creates a new user id for the writer program. It also modifies the files so that they are owned by the new user. Finally, it inserts setuid call in the writer program.

For Java programs, policy files are automatically created for programs running in a JVM.

Towards a Least Privilege Tool

strace(1) command is used in the example to analyze a binary file for its resource dependencies. Similar tools exist for other platforms, e.g. PE.Explorer in Windows. These can be reused by a Least Privilege tool. After the resources are identified, simple scripts can create users and groups, modify file ownership, and generate policy files.

---

2These steps are also part of the chroot Jail transformation described in detail the next chapter.
6.2 Securing System Perimeter

An application’s perimeter is the first line of defense. Many exploits can be prevented if proper authentication, authorization and input validation schemes are applied at the perimeter. It will be easier to apply the schemes if an application has few access points. Minimizing the access points of a system is a fundamental transformation, since it lays the foundation for other security features.

An additional benefit for isolating access points is privilege separation. Access points are more vulnerable to external attacks than internal parts of the system. Separating them into a gatekeeper component means that the internal parts of a system can run with a different privilege level. Thus an attacker cannot compromise the internal parts of a system after compromising an access point.

This section describes 2 transformations. *Single Access Point* transformation minimizes the access points of a system by extracting the access points and unifying them under a gatekeeper component. *Policy Enforcement Point* transformation adds a policy enforcement point to the system perimeter. Components for authentication, authorization and input validation are later added to the policy enforcement point. These program transformations are described in chapters 7, 8 and 9.

6.2.1 *Single Access Point* Transformation

You have a system with multiple access points. You want to secure the system from outside intrusion.

Minimize the number of access points.
Motivation

Having many access points means that a developer has to worry about securing many points and include authentication, authorization and input validation components at each of these points. Minimizing the number of access points makes a system simple, secure and easy to maintain.

Merging multiple access points creates a gatekeeper component where authentication, authorization and input validation components can be added. An example of a gatekeeper component is the SMTP service component (smap and smapd) of TIS Firewall Toolkit (FWTK), that acts as a gatekeeper for a sendmail program running in the background. The goal of smap and smapd is to preserve the functionality of sendmail, while preventing an arbitrary user from communicating directly with sendmail.

Preconditions

A program with the following characteristics benefits from a Single Access Point transformation.

- The program has multiple access points.
- Access logic is mixed with internal business logic of the application.
- The program is modular, i.e. the access functionality are in separate functions of modules. Access functionality can be extracted from internal parts.

Mechanics

A developer specifies the access points.

The program transformation extracts access functions in separate components. The extracted gatekeeper component(s) operate remotely from internal components. A thin unification layer is created between gatekeeper component(s) and internal components. The layer is a created by making a Façade [74] in the object-oriented world or by introducing a wrapper component that has the same API. The unification layer minimizes the access points and stages subsequent program transformations, typically Policy Enforcement Point transformation(section 6.2.2), Add Authentication Enforcer transformation(section 8.3.1), Add Authorization Enforcer transformation(section 9.3.1), Add Perimeter Filter transformation(section 7.3.1) etc.

The hardest part for applying this transformation is to determine and specify where the access points of a program are. For an application communicating with a socket, this will be the functions that bind and
listen to a socket. The tool section of the Partitioning transformation describes some program analysis tools that create control flow graphs of a program. These can be helpful in determining the access points.

Example

We applied the Single Access Point transformation to extract the access points in jftpd version 0.3, a Java FTP server. It contains 11 Java files with 2138 lines of code. jftpd runs everything in the same virtual machine. We created a gatekeeper component to run it in a separate virtual machine. After the extraction, the components communicate using RMI. We have done the entire process using Eclipse IDE. We manually performed the steps; one can easily visualize that an automated tool can be built to perform these tasks.

There are three main classes: Server, ServerPI and ServerDTP. Server class binds to a socket, and listens to client requests. When a request comes, it creates a ServerPI thread. ServerPI method `clientloop` parses incoming requests and processes them. It contains several handler methods, such as handle_user, handle_pass etc. The handler methods call methods in ServerDTP. ServerDTP processes the tasks, e.g. `sendFile` method processes RETR command submitted by handle_retr, `receiveFile` method processes STOR command submitted by handle_stor, etc. Figure 6.1 shows a UML class diagram showing the main classes and some relevant functions.

![Figure 6.1: Class diagram showing important classes and methods of jftpd](image_url)

The extraction of the gatekeeper component is an instance of partitioning. The most efficient partitioning has to minimize inter-component communications. For jftpd, Server and ServerDTP falls into separate partitions: suppose we call them gatekeeper partition and processor partition correspondingly. ServerPI is split between the partitions. The parsing part is in the gatekeeper partition, while the handler methods is a part of the processor partition. Thus, the only inter-component communication is the ServerPI parser (clientloop) calling the appropriate ServerPI handler method. This is done using RMI. Figure 6.2 shows the partition.
Figure 6.2: Class diagram showing the partitions

We manually made the changes in two steps. In the first step, we created the partitions by creating a separate package namespace called gatekeeper and moved all the parts that will eventually be in the gatekeeper partition in this package. Moving class to a package is a refactoring, as well as splitting a class (Extract Class refactoring). Eclipse provides tool support for these.

Then we replaced the local calls in ServerPI in the gatekeeper partition with RMI calls. This was done in three steps.

1. Creating an RMI interface describing the handler methods in ServerPI in the processor partition.

2. Convert ServerPI class in the gatekeeper partition so that its local calls are replaced with RMI calls described by the interface.

3. Convert ServerPI class in the processor partition so that it implements the RMI interface. This might involve adding a stub.

The first step is an instance of Extract Interface refactoring. The second and third steps are not direct instances of refactoring, but they can be automated easily. In fact, Java RMI framework automates tasks such as stub generation.

Towards a Single Access Point Tool

The steps of extracting a gatekeeper component can be done with existing refactoring tools. Creating communication between the gatekeeper components and internal components can also be automated. Program analysis and visualization tools can be used to help a developer determine the access points of an application. Single Access Point transformation tools reuse Partitioning tools for extraction. We have discussed how Partitioning tools can be built in section 6.1.1.
6.2.2 Policy Enforcement Point Transformation

You want to centralize authentication, authorization and input validation at the system entry point.

Create a policy enforcement point component that takes authentication, authorization and input validation decisions.

Motivation

When a system has minimal number of access points, then authentication, authorization and input validation components can be added. But if these components directly communicate with the access points, the points become complex. To simplify the architecture of access points, a separate policy enforcement point should be created that communicates with the access points, as well as the authentication, authorization and input validation components. The policy enforcement point acts like a Mediator [74].

A Policy Enforcement Point transformation is dependent on a Single Access Point transformation. A Single Access Point transformation minimizes the number of access points. A Policy Enforcement Point transformation adds a component that stages authentication, authorization and input validation components.

Preconditions

A program with the following characteristics benefits from a Policy Enforcement Point transformation.

- The program has a minimal number of access points.
- A developer wants to add authentication, authorization and/or input validation functionalities at system perimeter.

Mechanics

A developer specifies the system entry points.

The transformation creates a policy enforcement component and placeholder components for authentication, authorization and input validation. The access points delegate incoming requests to the policy enforcement point component. The placeholder components for authentication, authorization and input validation are replaced by the subsequent program transformations, and these components take the actual
authentication, authorization or input validation decisions. A policy enforcement point component simply mediates between all these components.

Example

Suppose, an imaginary developer Alice has a Java program that connects to a hostname and a port via a socket. It connects to two hostname/port pairs and uses the data received from one connection to query an SQL database and the data received from another connection to query an LDAP database. Alice wants to retrofit authentication, authorization, and input validation components to the program.

Alice first minimizes the access points by applying Single Access Point transformation (section 6.2.1). This transformation is followed by a Policy Enforcement Point transformation. A policy enforcement point provides a single check point for authentication, authorization and input validation.

Suppose, Alice extracts the access points to a new AccessPoint class. Alice selects the class and applies a Policy Enforcement Point transformation. Figure 6.3 shows the classes added.

![Figure 6.3: Classes added by a Policy Enforcement Point transformation](image)

The PolicyEnforcementPoint class delegates authentication request to a class that handles the request (AuthenticationEnforcer). This placeholder class is the starting point of the next transformation that creates a username/password-based authentication component (Add Authentication Enforcer transformation described in section 8.3.1). Similarly, placeholder classes are added for authorization and perimeter filter components.
Towards a Policy Enforcement Point Tool

This transformation is easily automated, since it only adds a policy enforcement point class. The place holders for authentication, authorization and input validation components are filled by corresponding transformations, e.g. *Add Authorization Enforcer* transformation(section 9.3.1) and *Add Perimeter Filter* transformation(section 7.3.1).

### 6.3 Organizing the Transformations

Fundamental program transformations are organized using two criteria: their impact on codebase and the type of transformation. Table 6.1 lists the program transformations in this group.

<table>
<thead>
<tr>
<th></th>
<th>Small, infrequent change</th>
<th>Small, frequent change</th>
<th>Large change</th>
<th>Change beyond boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Policy Enforcement Point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distribute</td>
<td></td>
<td>Partitioning</td>
<td></td>
<td>Single Access Point</td>
</tr>
<tr>
<td>Limit</td>
<td>Least Privilege</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Fundamental Security-oriented Program Transformations

*Partitioning* program transformation accepts a component and a specification of functional distribution from developers and *distributes* the tasks into multiple logical partitions so that each partition can run with a single privilege level. It is an architectural level transformation that has a *large impact*. Large transformations are harder to automate and they require more detailed specification from the developers. For a *Partitioning* transformation, a developer has to specify the functionality distribution, interface, privilege level and remote communication mechanism.

The minimum privilege required for a process can be automatically determined with a *Least Privilege* transformation. This makes *small, frequent change* to the source code; most of the modification goes into modifying the environment of a process once the analysis step returns a privilege level. It sets the *limit* for a process; the limit is the just enough privilege.

A *Single Access Point* transformation centralizes the access points of a system by redistributing them. This program transformation has a *large impact*. The implementation mechanism of this transformation is similar to a *Partitioning* transformation.
After a Single Access Point transformation is applied to minimize the access points, a Policy Enforcement Point transformation is applied to create a separate component. A Policy Enforcement Point transformation has a small impact, and it only applies at a few program points. It can be classified in the add component category.

6.4 Composing the Transformations

The two transformations for privilege separation follow a strict order. An application should be partitioned before applying the Least Privilege transformation. Reversing this order does not make sense, because a monolithic application will always run with a higher privilege than a carefully partitioned application. In reality, these two program transformations go hand in hand. The privilege level determined by the Least Privilege transformation guides how an application should be partitioned. A developer can 1) apply Partitioning, 2) check the privilege level, then redo steps 1 and 2 multiple times before finding the partitioning sweet spot.

Similarly, a Single Access Point should strictly be applied before a Policy Enforcement Point transformation. Otherwise, multiple authentication, authorization and input validation logic are distributed in multiple places of an application. This makes them harder to maintain.

6.5 Summary

This chapter describes 4 fundamental program transformations that precede other transformations. They secure a system’s perimeter as well as its internal parts.
Chapter 7

Security-oriented Program Transformations for Input Validation

In November 1988, Morris worm, the first of its kind, was released in the Internet [1]. The worm abused the debug function of the sendmail program [63]. sendmail has a feature to send mails to a program, such that the program receiving the mail executes with the body of the mail message as input parameters. sendmail allows this only in the debugging mode. Unfortunately, the sendmail program packaged with 4.3BSD and pre-4.1 versions of SunOS had the debugging mode turned on by default. The worm used this feature to connect to a sendmail daemon and send a message to a shell program of the recipient. It then passed the body of the message as a parameter to the shell. The body consisted of a bootstrap shell script that, when executed, would connect back to the attacking machine via TCP and download a pre-compiled object code replica of the worm. It then tried to link the target code and execute it. At this point, the machine became infected. The worm then sent the same mail message from the infected machine to other sendmail servers.

Every program is designed with an input domain that it can process without error; it is as if a program has a comfort zone [171] of inputs. The comfort zone is determined by the designers, perhaps using the requirements of a program as a guideline. The concept of a comfort zone is broad. The comfort zone for a program can be as fine grained as an assurance of the type safety of inputs, or as coarse grained as the requirement that a mail transfer agent program should treat its inputs as mail messages only, and not treat them as inputs to another program.

![Comfort Zone Diagram](image)

Figure 7.1: Comfort zone of inputs and input rectification
Figure 7.1 shows the concept of comfort zone. A program should only process inputs inside the comfort zone, and discard inputs that are outside the zone; but incorrectly designed programs that allow inputs outside the comfort zone are all too common. Some of these inputs are non-malicious; at the worst they crash a system\(^1\). But some inputs are malicious; an attacker can pass these inputs to a program and then take control of the entire system. Unvalidated inputs in programs allow attackers to launch data injection attacks.

The typical solution for preventing unvalidated input vulnerability is to insert filters that check and validate input variables. The validation process can check an input variable so that it is inside the comfort zone, or attempt to rectify a variable, i.e. modify the input variable so that it maps inside the comfort zone (or at least, non-malicious input zone). Figure 7.1 shows the rectification process.

This chapter describes 10 security-oriented program transformations to remove unvalidated input vulnerabilities from programs. These transformations follow various solution strategies, some add perimeter filters that validate incoming data, others validate specific input variables, while others protect file system. Some program transformations introduce testing methods so that the developers do not have to manually write input validation tests. The transformations in this chapter introduce filters to validate inputs coming from external sources. There can also be logical errors generated by internal inputs; some program transformations can validate internal inputs, but these inputs are mostly out of scope.

The next sections narrate the input validation problem and describe these program transformations. This is followed by sections covering how these program transformations are organized and how can they be applied together.

### 7.1 Unvalidated Input Vulnerability

OWASP [151] lists 18 variants of data injection attacks that target different programming environments, e.g. SQL injection attack affects a database access language, LDAP injection attack affects a directory access protocol, cross site scripting (XSS) injects malicious code through an HTTP payload, etc. Buffer overflow attack and format string attack are also variants of injection attacks. Overall, injection attacks are the source of most attacks, posing the most insidious threat to modern software.

\(^1\)That saying, denial of service can be a serious security problem. Chapter 11 describes how program transformations can be applied to prevent denial of service attacks.
The mechanism of injection attacks are similar: passing invalid data when valid data is expected. But the intent of launching an injection attack can be different. We have distinguished 3 intents.

1. The most common intent is to trick a program to reveal unauthorized information. An example is an attacker launching a blind SQL injection attack. An attacker passes SQL code where data is expected and the resulting malformed SQL statement reveals unauthorized information from a database. Similarly, a cross site scripting attacker tricks a program to reveal cookies.

2. The second intent is to modify the execution environment so that a program runs in an unexpected manner. In a buffer overflow attack or a format string attack, an attacker corrupts the stack or heap. In a command injection attack, a program is tricked to execute arbitrary commands.

3. The third intent is to directly corrupt resources, e.g. files. Multiple writer processes writing to a shared file can corrupt the file if an attacker manages to crash a process at the wrong time.

These three intents are not disjoint: the same attack can be launched to attain three separate goals. An example is a ‘time of check to time of use’ (TOCTTOU) race condition attack. Unlike other injection attacks, an attacker does not pass a malformed input to a program. Instead, it modifies the program environment so that the binding of a name to an object changes between repeated references [20]. For example, in an access(2)/open(2) race condition attack, a program checks whether a file exists and then opens it, but an attacker replaces the checked file so that the program opens a new, malformed file. The attacker can do it for many reasons. He can use it to read a file that he otherwise does not have read access (reveal unauthorized information). Or he can modify a configuration file with his own version (run a program in an unexpected manner). Also he can delete or corrupt important system files (corrupt resources).

7.2 Solution Strategies

Current works on data injection attacks focus on three broad classes of attacks: SQL injection, cross site scripting, and buffer overflow. Many sophisticated static and dynamic analysis tools are available for automatically detecting instances of these data injection vulnerabilities in the source code. Most of these

2We will discuss the static and dynamic analysis tools in the description of the relevant program transformations.
solutions stop one step short of solving the problem, because they still require a programmer to manually add security checks in the program to prevent the attack. People are bad at repetitive manual tasks – computers are much better.

We wanted to find out whether a program transformation based approach is useful for programmers. We surveyed the database on several random weeks to find out whether the vulnerabilities have common causes that can be repaired by program transformations. For example, on the first week of September 2008, Bugtraq listed 28 instances of buffer overflow vulnerabilities affecting software from 22 different vendors. At least 17 of these incidents occur due to the use of unsafe functions and the common fix is to validate the input or replace the unsafe function. Despite the common problem and solution, there are no general purpose tools that these developers can use. Also, a program transformation based solution often attempts to fix all instances of a vulnerability instead of fixing the vulnerability that has been reported. This will prevent the same bug from resurfacing in another part of a program. For example, Bugtraq lists 5 instances of buffer overflow vulnerability in different versions of sendmail; it also lists 32 instances of various data injection vulnerabilities in Microsoft Word 2000. Each time developers created a patch to remove the specific vulnerability, instead of attempting to remove the vulnerability from the application.

The 10 program transformations described in this chapter follow four solution strategies to combat injection attacks.

1. The first line of defense is to add a series of filters at the system perimeter. We describe 2 program transformations. Add Perimeter Filter and Message Intercepting Gateway transformations add input validation filters at the system perimeter, following different implementation strategies.

2. We describe 5 program transformations to validate input variables before use. A Decorated Filter transformation adds input validation filters. Safe Library Replacement and Safe Type Replacement transformations remove unsafe functions and datatypes. An Explicit Type Enforcement transformation explicitly adds type information to prevent datatypes to be used in an unsafe context. A Randomization transformation does not validate explicitly; but it makes common attack vectors ineffective by varying the application itself.

3. A somewhat different strategy is to limit an application’s ability to touch and corrupt files in a file

---

3We can analyze 19 incidents out of 28, since the remaining incidents were reported about proprietary software for which the source code was unavailable.
Two program transformations, *chroot Jail* and *Unique Location for Each Write Request*, prevent file system corruption.

4. Finally a program transformation can automatically add testing mechanism. A *Fuzzing* transformation adds fuzzing tools to test a program’s input domain.

The next four sections describe these program transformations.

### 7.3 Adding Filter Components at System Perimeter

#### 7.3.1 Add Perimeter Filter Transformation

You want to scan and check incoming data for malicious content before it is used in the system. *Add a component at the system entry point to validate data.*

**Motivation**

For every application, there are some input validation policies that are common for all inputs. Typically these input validation policies do not depend on internal business logic and can be applied to validate all incoming inputs. For example, an incoming email message can be validated against a network address blacklist to ensure that the message is not originating from a server known for spamming. Another example is an HTTP server checking that the incoming inputs contain a unique session key to prevent cross-site request forgery attacks. It is better to enforce these policies at the perimeter so that the internal parts can concentrate on input-specific policies. Besides, writing checks for these policies for every input variable duplicates a lot of code. One way of eliminating duplication is to move the checks at the application entry point.

*Single Access Point* (section 6.2.1) and a *Policy Enforcement Point* (section 6.2.2) transformations minimize the number of access points, and create a policy enforcement point that stages authentication, authorization and input validation components. This transformation adds a filter component, typically replacing the placeholder filter component added by a policy enforcement point transformation. Alternatively it can create a chain of filter components.
Pre-condition

A program with the following characteristics benefits from this transformation.

- There are systemwide policies applicable to all incoming data and the policies can be enforced without knowledge of the business logic.

- The program has a few entry points. Before applying this transformation, a developer can apply a \textit{Single Access Point} transformation to minimize the number of entry points (section 6.2.1).

Mechanics

A developer specifies where to insert a perimeter filter and what policies to apply.

The program transformation adds a component that contains a collection of filters. The filters implement the input validation policies specified by the user [194]. The program transformation adds a request at the entry point to delegate input validation to the secure base action component which then passes the input through the filters to validate the input.

Example

Figure 7.2 shows the transformation applied to a Java program. A developer specifies the insertion point which, in this example, is a JSP or Servlet front controller [5]. A developer also specifies a list of policies.

![Diagram of Add Perimeter Filter transformation]

Figure 7.2: Applying \textit{Add Perimeter Filter} transformation

The transformation creates the SecureBaseAction class as a policy enforcement point. It also creates an InterceptingFilter class that contains the Filters. All the incoming user inputs are passed to SecureBaseAction, which validates by passing inputs through a series of filters. Figure 7.3 shows the rectification process.
Towards a Perimeter Filter Transformation Tool

A special tool is not needed to perform tasks like adding a new class, creating a superclass and delegating tasks. These are examples of refactorings [71] for which tools are already available in many programming platforms.

The hardest part for a user is to determine the perimeter of a program. For web applications, the perimeters are well-defined; but for other applications, the perimeters may be less clear. For example, an FTP server listens to port 21 before authentication, and then switches to port 20. In this case, the application will have two entry points, both of which need to be covered. Applying Single Access Point and Policy Enforcement Point transformations before this transformation creates well-defined perimeters; nevertheless, users need to have deep understanding of program behavior to apply them. We have described some program analysis tools that can be helpful in guiding users in the description of Single Access Point (section 6.2.1) and Partitioning (section 6.1.1) transformations.

A program transformation tool should create a collection of filters according to user specification. The tool should contain a library of filters that can be used directly or customized using parameters. A developer should also be able to extend the library. A discussion on various types of policies that can be applied is in the tools section of Decorated Filter (section 7.4.1) transformation.

The implementation of an Add Perimeter Filter tool for a non object-object oriented program does not use inheritance. There the incoming request is delegated to a newly created perimeter filter component. Multiple perimeter filter components, each implementing a group of specific type of filters, can be chained together. Each delegates the output to the next component in a pipes and filters [35] architectural style.
7.3.2 *Message Intercepting Gateway Transformation*

You have a specification available that describes the perimeter interface of an application. You want to scan and check incoming and outgoing messages for malicious content.

*Create a wrapper component from the perimeter specification. The wrapper component acts as a firewall; it intercepts and validates all incoming data before forwarding it to the internal parts of an application.*

![Diagram](image)

**Motivation**

An *Add Perimeter Filter* transformation (section 7.3.1) creates a new component containing a collection of input filters; the main program (or a policy enforcement point component) delegates incoming data to the perimeter filter component. An alternative way of implementing filters is to have a wrapper component that intercepts requests. This can be used in Web services to validate incoming data. A Web service interface is described by a WSDL specification; a wrapper component can be built from this specification.

**Preconditions**

A program with the following characteristics benefits from a *Message Intercepting Gateway* transformation.

1. The program is a Web service with WSDL specification.
2. There are systemwide policies applicable to all incoming data and the policies can be enforced without knowledge of the business logic.

**Mechanism**

A developer specifies the location of a WSDL file for a service and what policies to apply.

The program transformation creates a wrapper component from the WSDL specification. The wrapper component acts as a proxy. It delegates the requests to the services, but only after validating the input with a set of filters. The typical, simplest implementation of a message interceptor gateway only checks incoming XML data and validates it. This can involve validating the encrypted or signed messages, checking the integrity with hash value etc.

**Example**

Figure 7.4 shows how a typical message interceptor gateway works. The incoming message is intercepted and passed through a series of filters (Message Handler Chain in the figure). It can also communicate with an authentication enforcer component and authenticate based on provided information. After the validation steps, data is presented to the service endpoint.

![Sequence diagram showing how a message interceptor gateway works](image)

Figure 7.4: Sequence diagram showing how a message interceptor gateway works

The program transformation creates the message interceptor gateway and the chain of filters based on a developer’s selection.
Towards a Message Intercepting Gateway Tool

There are several available tools for generating Web service front ends automatically from a WSDL specification, e.g. wsdl.exe in Visual Studio .NET to create a front end in C#, VB, or VJ#, or WSDL2Java from Apache Axis to create a front end in Java. Once the front end is created the code follows a template that delegates the call after passing it through a series of filters.

Other than policies for validating data, several other checks can be done at the system entry point. For example, an XML message can be verified for well-formedness according to an XML schema, or compliance with standards such as OASIS WS-Security and SAML token profile. Also, messages can be verified for XML digital signature and XML encryption. It is possible to compose framework specific classes to create these validation filters.

7.4 Validating Input

7.4.1 Decorated Filter Transformation

You want to apply multiple policies to a program variable that holds user input.

Validate an input variable by decorating it with a series of filters implementing the policies.
Motivation

In order to prevent input validation attacks, programmers manually write checks to validate inputs. They determine a comfort zone [171] of inputs and write checks to prevent inputs outside the zone. Mistakes in writing checks is very common. In one random week (the first week of January 2009), Bugtraq lists 41 SQL injection and 13 XSS attack instances that can be solved by more stringent input validation.

Replacing manual checking with automated tools increases programmer efficiency because they can concentrate on policies rather than the mechanism of implementing checks.

Preconditions

This transformation applies to object-oriented programs. A program with the following characteristics benefits from a Decorated Filter transformation.

1. The program has injection vulnerabilities originating from unsafe inputs. Inputs are incompletely or incorrectly checked.

2. There are known attack patterns. These can be used as a basis to implement the validation policies.

3. Rectification policies should not transform valid inputs.

Mechanics

A developer specifies the target input variable and the policies to be applied.

The program transformation adds the policies to an input variable by using Decorators [74]. Figure 7.5 describes how a string variable is decorated with policies that remove SQL injection attack vectors. The string input variable becomes encapsulated in the abstract AbstractStringContainer class. Its concrete instance UnsafeString becomes the target of input validation policies.

Example

We have written a proof-of-concept Eclipse plugin to apply SQL injection prevention policies to Java programs. A programmer specifies the variable to validate as well as the policies. We illustrate the transformation tool by applying it on an insecure program.
An insecure program instance. Class `DBConnect` contains a method for querying a database and showing the result (`showData`). The `showData` method reads input from standard input (line 7), and prepares the query and executes it (lines 9–16). The database contains a single table named `users`, with three fields for storing user id, user name and password.

```
import java.sql.*;

public class DBConnect {
  ...
  public void showData() {
    ...
    String username = stdin.readLine();
    ...
    try {
      stmt = connection.createStatement();
      resultSet = stmt.executeQuery("select * from users " +
      "where username = '' or '1'='1' + username + ");
      } catch (SQLException e) {
        e.printStackTrace();
      }
  }
}
```

Attacking this program is straightforward. A malicious user enters as input the string, 

' or '1'='1.

The resulting query is,

```
select * from users where username = '' or '1'='1'
```

Applying the transformation. In the example program, we have applied policies to remove AND and OR from user inputs (see figure 7.5).
Towards a Decorated Filter Tool

A typical tool for a Decorated Filter transformation should be interactive, similar to refactoring tools. Let us describe a typical scenario. Suppose, a developer wants to apply the program transformation on a Java program opened in Eclipse IDE. The program transformation can be available as part of the refactoring catalog. The developer selects a program variable (e.g. username in the program in the Example section) and invoke the transformation. The transformation asks the user to select from a set of filters. Then the tool adds the selected filters by applying the program transformation.

Thus, a tool for Decorated Filter transformation consists of two parts: there is the part that applies the filters to existing code, and there is also a part that contains a library of filters for various types of vulnerabilities.

Applying filters as decorators is an instance of Moving Embellishment to Decorator [116] refactoring.

A useful tool should present a lot of different types of policies that a developer can choose from. The tool should come with a built-in policy library. Moreover, the tool should be extensible, so that developers can add their own custom policies and apply them using a Decorated Filter transformation tool.

Add Perimeter Filter (section 7.3.1) and Decorated Filter transformations apply various input validation
policies to input variables. The policies differ for different injection attacks. Table 7.1 lists some attacks and corresponding input validation policies.

<table>
<thead>
<tr>
<th>Type of Injection Attack</th>
<th>Example Policy</th>
<th>Description of Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Injection</td>
<td>1. Remove SQL keyword</td>
<td>Searches input for SQL keyword, e.g. SELECT, AND, OR, UNION etc. and removes the keyword. Before: <code>' OR '1'='1</code> After: <code>'1'='1</code></td>
</tr>
<tr>
<td></td>
<td>2. Remove/Strip single quote</td>
<td>Searches input for single quote and remove/strip it. Before: <code>' OR '1'='1</code> After: OR 1=1 (Remove) After: (Entire string is stripped)</td>
</tr>
<tr>
<td></td>
<td>3. Escape single quote</td>
<td>Searches input for single quote characters and escapes them. Before: <code>' OR '1'='1</code> After: <code>' OR '1'='1</code></td>
</tr>
<tr>
<td></td>
<td>4. Remove comments</td>
<td>Searches input for SQL comments and removes comments Before: <code>' OR '1'='1</code> - something else After: <code>' OR '1'='1</code></td>
</tr>
<tr>
<td></td>
<td>5. Decode hex encoding</td>
<td>Decodes any part of the input that has been encoded in hex. Before: 0x73656c656374 After: SELECT</td>
</tr>
<tr>
<td>Direct Static Code Injection</td>
<td>6. Remove commands</td>
<td>Searches input for system commands such as system, rm etc. Before: rm%20-rf%20/ After: %20-rf%20/</td>
</tr>
<tr>
<td></td>
<td>7. Remove &amp; and</td>
<td>Searches and removes LDAP and (&amp;) and or (</td>
</tr>
<tr>
<td>Log Injection</td>
<td>9. Remove &lt;SCRIPT&gt; tag</td>
<td>Searches for <code>&lt;SCRIPT&gt;</code>...<code>&lt; /SCRIPT &gt;</code> and removes it. Before: <code>&lt;SCRIPT&gt;alert('XSS');&lt;/SCRIPT&gt;</code> After: <code>&lt;SCRIPT&gt;</code> tag removed</td>
</tr>
</tbody>
</table>

Table 7.1: Sample policies for various transformations

Most of the policies listed in Table 7.1 validate input by rectifying it. The goal of all injections is to run code in place of data; hence most policies remove/replace keywords and special characters. These policies derive from catalogs of attack patterns. For example, the rectification policies for SQL injection attacks derive from a comprehensive list of SQL injection attack patterns [189, 130, 158]. The policies for preventing XSS attacks derive from a comprehensive list of attack patterns [178].

The policies in Table 7.1 searches and removes attack vectors. An opposite approach is to have policies that describe the expected input, and discard input if they fail to comply with that description. For example, a user can specify that the maximum length of a string datatype is 20 and it contains alphanumeric...
There are some policies that modify input, but not for removing attack vectors. These target various filter evading patterns. Attackers try to evade the filters by encoding inputs. In response, there are some checks that decode inputs and convert them to a canonical format. In Table 7.1, policies 5 and 11 do so.

Another type of policy augments input with metadata. Such policies, applied at the system entry point, mark the input and then use it for taint based injection attack detection. WASP [97] augments a Java string with a MetaString class to detect SQL injection; SQLCheck [197] adds marks to the beginning and end of a Java string to detect various types of command injection; Pixy [113] adds metadata to PHP strings to detect cross site scripting. These tools apply policies to data twice–1) they add metadata to an input at system entry point, and 2) they check the metadata to detect a malicious activity before the input is used.

Other policies apply static and dynamic analysis techniques to build a model of good input and match it with the actual input before it is used. AMNESIA [96] statically builds an NFA to model SQL statements, CANDID [12] models SQL statements with a parse tree. These policies decide whether an input is good or malicious.

Not all policies are for input validation. Encryption, encoding, parity checking, etc. can also be thought as special policies applied on data. Program transformation tools that decorate an input with filters can automate these tasks.

Some policies can be odd; they replace valid data with fabricated data. An example is a policy that replaces error messages with a generic error message so as to not reveal any internal information. Another odd thing about this policy: it is applied to output data rather than input.

The use of policies is a property of program transformations that distinguishes them from refactoring. Although both take user specification and make structural changes, a refactoring only needs to know the abstract syntax tree of a program to make the change [172]. In contrast, a security-oriented program transformation needs to additionally know the artefacts that make the behavioral change. For example, a Decorated Filter transformation needs to know about the library of filters to use to change program behavior.

A Decorated Filter transformation may include input validation policies that must be applied in a particular order. The policies that canonicalize input are applied before any other policies. Some other policies may keep the order information. For example, if an input is encrypted before the calculation of its message
digest, it is to follow a reverse order when it is decrypted. An automated tool for the transformation should encapsulate these structural details.

As new kinds of security threats continue to appear, even systems that meet high security standards today will eventually require updates; they will need security-oriented program transformations. To cope with requirement changes, organizations will have to modify existing policies or create new ones. Thus policies used by a program transformation should be parameterizable and extensible.

### 7.4.2 Explicit Type Enforcement Transformation

You have a program in which arithmetic operations have operands with different types; the end result may contain an unexpected value.

*Explicitly declare or cast the type of operands so that they are properly handled in an operation.*

**Motivation**

When arithmetic operations mix the type of operands, the result may be unexpected. For example, a signedness bug occurs when an unsigned variable is interpreted as signed, or when a signed variable is interpreted as unsigned [22]. This type of behavior can happen because internally to the computer, there is no distinction between the way signed and unsigned variables are stored. Hence, the end result is not the one originally intended. If an unsigned integer is compared to a signed integer in a program, an attacker can carefully inject inputs to bypass the comparison. Similarly, if signed values are used in an arithmetic operation, an attacker may cause an overflow and store a value with a wrong sign. When this is used in an unsigned context, an error will occur.

If a programmer explicitly mentions the type of numeric operands, a compiler can unambiguously enforce the type in an arithmetic operation. Otherwise, it may promote numeric operands unsafely, causing an integer overflow.

**Preconditions**

A program with the following characteristics benefits from an *Explicit Type Enforcement* transformation
1. The program uses numeric variables in an unsafe manner, e.g. it uses unsigned values in a signed context and vice versa.

2. The program gets integer input from end users, and use them. The user inputs can be used to create an integer overflow.

Mechanism

A developer specifies the target input variable.

The program transformation checks whether the input variable has been used in an unsafe context. It explicitly casts the type of variables in an arithmetic operation, or it changes the type of the variables used in an operation.

Example

A programmer specifies the input variable to validate. We illustrate the transformation tool by applying it on an insecure program.

The following C program shows an unsafe situation that an attacker can exploit. The program copies to a buffer buf in line 11. The buffer is of size BUF_SIZE. Line 10 compares two integers so that at most BUF_SIZE bytes are written.

```c
#include <stdio.h>
#include <stdlib.h>

#define BUF_SIZE 10

void main (int argc, char * argv[]){
    int size;
    char buf[BUF_SIZE];
    size = atoi(argv[1]);
    if (size <= BUF_SIZE){
        memcpy(buf, argv[2], size);
    }
    ...
    ...
}
```

An attacker can introduce exploit the program and eventually launch a buffer overflow attack. This is because the signed integer variable size is compared against BUF_SIZE in line 10. Then the signed integer variable is used in an unsigned context in the function memcpy in line 11. If an attacker passes a negative
integer, he or she will satisfy the check in line 10. But then the value will be converted to a large unsigned number in line 11. Consequently, the attacker will be able to overflow the buffer and inject a payload that he is passing from the command line. A carefully crafted payload can cause a buffer overflow attack, and can allow the attacker to take control of the system.

A developer can apply an *Explicit Type Enforcement* transformation on the size variable. The program transformation can make two changes. It identifies that a signed integer has been used in an unsigned context and modifies the type declaration in line 7.

```c
#include <stdio.h>
#include <stdlib.h>
#define BUF_SIZE 10
void main (int argc, char * argv[]){
    unsigned int size;
    char buf[BUF_SIZE];
    size = atoi(argv[1]);
    if (size <= BUF_SIZE){
        memcpy(buf, argv[2], size);
    }
    ...
}
```

Alternatively, it can retain the variable declaration and explicitly force typecasting in all the places the variable size is used.

```c
#include <stdio.h>
#include <stdlib.h>
#define BUF_SIZE 10
void main (int argc, char * argv[]){
    int size;
    char buf[BUF_SIZE];
    size = atoi(argv[1]);
    if ((unsigned) size <= BUF_SIZE){
        memcpy(buf, argv[2], size);
    }
    ...
}
```

In line 10, the comparison mixes a signed integer with an unsigned integer. By explicitly promoting the type in the comparison operation, the program transformation ensures that an attacker cannot bypass the operation by passing a signed value.
Towards an Explicit Type Enforcement Tool

A typical tool for an Explicit Type Enforcement transformation should be interactive, similar to refactoring tools. Suppose a developer wants to apply the program transformation on a C program opened in Eclipse IDE. He or she selects a variable in the program and apply the transformation. The transformation enforces the type so that program is not vulnerable to integer overflows.

A Explicit Type Enforcement transformation tool applies dataflow analysis and taint analysis techniques to enforce type for numeric variables. The program transformation follows the MISRA C [142] guidelines to ensure that unsafe type conversions cannot occur.

Using taint analysis to detect integer misuse vulnerabilities in C programs has proven to be highly effective. There are several program analysis techniques to detect integer overflow vulnerabilities in C programs that can be used for program analysis. Ceesay et al. [38] implemented a static analysis tool on top of Cqual [70] to track the untrusted data. They annotate data with type qualifiers. The qualifiers categorize integer variables into two types: trusted and untrusted. If an untrusted integer variable is used to access memory, an alarm is reported. Sarkar et. al. [181] uses a constraint graph that detects whether a tainted data has been sufficiently validated. Ashcraft et. al. [9] presents a range checker to track whether an untrusted value is bounds checked before the value is used in trusted sinks.

Smartfuzz [57] uses symbolic execution to construct test cases that trigger arithmetic overflows, non-value-preserving width conversions, or dangerous signed/unsigned conversions. Then, they run the program on these test cases and use standard tools that check for buggy behavior to recognize bugs. SmartFuzz can be used for dataflow analysis and detect potential integer overflows.

There are other tools and compiler extensions that target integer overflow errors. There are compiler extensions to prevent integer based vulnerabilities. With -ftrapv option, the gcc compiler inserts additional calls (e.g., _addvsi3) before signed addition operations to catch overflow errors. There are several integer overflow detection tools that apply on binaries [217, 215]. The internal mechanism of these tools can provide useful information for developers who want to develop a program transformation tool.
7.4.3 Safe Library Replacement Transformation

You have a program that uses a function that may cause data injection attacks if it receives an insufficiently validated input. You want to ensure that the program is not vulnerable to injection attacks.

*Replace unsafe functions with safe functions that are not vulnerable even if malicious data is injected.*

Motivation

In one random week (the first week of September 2008), Bugtraq listed 28 buffer overflow vulnerabilities out of a total of 154 vulnerabilities. We can not analyze 9 instances that affect proprietary software. 17 of the remaining 19 instances can be solved by replacing unsafe string functions with safe functions. In the remaining two cases, buffer overflow vulnerabilities originated from direct manipulation of pointers.

sendmail uses unsafe functions for string copy and concatenation. qmail prevents buffer overflow by using safe functions that check for the size of the destination buffer before making an update.

Many sophisticated static [209, 213] and dynamic [54, 99] analysis tools are available for detecting vulnerable functions that lead to buffer overflow attacks. To prevent buffer overflow, programmers write checks or use safe buffer handling functions that check buffer bounds before performing an operation. Some of these functions trim the resultant destination buffer to fit its size [108, 141], while other functions dynamically resize the destination buffer [135, 110].
Usually, programmers manually inspect code to make these changes. We asked the developers of the top ten most active projects of all time in sourceforge.net [191] about their development approach. Six projects use C or C++ as one of the languages, three use PHP and one uses Java. In five out of six C/C++ projects, programmers initially used unsafe `strcpy` and `strcat` functions, but manually changed to safer C/C++ string libraries later.

Manual changes are error-prone. This method does not scale for large projects. For example, Ghostscript (about 800 KLOC) is a medium size program, but its programmers have replaced only a small part of its unsafe functions with safe functions. A program transformation automatically replaces all instances of unsafe string functions.

**Preconditions**

A program with the following characteristics benefits from a library replacement transformation

1. The program uses unsafe functions for which safe alternatives are available.

2. The program does not have critical performance requirement. Safe functions are slower than unsafe functions because they internally perform various checks. Replacing functions for a real time system may have a bad impact.

**Mechanism**

For each unsafe function, a developer specifies the alternative safe function and the library that includes the function.

The program transformation finds all functions that need to be replaced. It replaces unsafe functions with suitable alternatives in all source files. It adds information about the new library to configuration files so that the new program compiles. Table 7.2 lists some unsafe functions that cause buffer overflow attacks in C/C++ and their safe alternatives.

Safe functions prohibit buffer overflow by trimming the destination buffer. This may cause a program to behave differently; but this happens only for attackers who try to pass long buffers. For example, a safe

---

4Ghostscript 8.64 has 681441 lines of code in .c and .cpp files and 110911 lines of code in .h files. It has 209 instances of `strcpy` and 109 instances of `strncpy` functions. It also has 166 instances of `strcat` and 6 instances of `strncat` functions. Ghostscript developers believe that the remaining uses of `strcpy` and `strcat` functions are safe.

---

99
<table>
<thead>
<tr>
<th>Unsafe string functions</th>
<th>Safe string functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcpy(3), strncpy(3) - Copy string</td>
<td>g_strlcpy from glib library [141]. astrcpy, atsnrcpy from libmib library [65],</td>
</tr>
<tr>
<td>char *strcpy (char *dst, const char *src); char *strncpy (char *dst, const char *src, size_t num);</td>
<td>StringCchCopy, StringCchCopyN, StringCchCopyEx from SafeCRT library [138]. safestr_copy and safestr_ncopy from SafeStr library [135].</td>
</tr>
<tr>
<td>strcat(3), strncat(3) - Concatenate string</td>
<td>g_strlcat from glib library [141]. astrcat, astrnacat from libmib library [65],</td>
</tr>
<tr>
<td>char *strcat (char *dst, const char *src); char *strncat (char *dst, const char *src, size_t num);</td>
<td>StringCchCat, StringCchCatN, StringCchCatEx from SafeStr [138], safestr_concatenate from SafeStr library [141].</td>
</tr>
<tr>
<td>gets(3) - Get input from stdin</td>
<td>gets(3) from ISO/IEC 24731 [109]. gets from SafeCRT library [127]. StringCchGgets, StringCchGgetsEx from SafeStr library [138].</td>
</tr>
<tr>
<td>char *gets (char *dst);</td>
<td>char *gets (char *dst, int dst_size, FILE *stream); char *afgets (char **dst address, FILE *stream);</td>
</tr>
<tr>
<td>memcpy(3) - Copy memory area</td>
<td>memcpy(3) from ISO/IEC 24731 [109]. memcpy from C99 [108]. memcpy from libmib library [65], gets_s from SafeCRT library [127].</td>
</tr>
<tr>
<td>void *memcpy (void *dst, const void *src, size_t num);</td>
<td>char *memcpy_s (void *dst, size_t dst_size, const void *src, size_t num);</td>
</tr>
<tr>
<td>getenv(3) - Get value of an environment variable</td>
<td>getenv(3) from ISO/IEC 24731 [109]. getenv from C99 [108]. getenv from libmib library [65], gets_s from SafeCRT library [127].</td>
</tr>
<tr>
<td>char *getenv (const char *name);</td>
<td>char *getenv (const char *name, size_t *return_value, size_t *dst_size, const char *name);</td>
</tr>
</tbody>
</table>

Table 7.2: Some unsafe functions and their safe alternatives

mail transfer agent program receiving a mail with a long header or body will truncate the data and produce garbage mail.

Safe library replacement transformation is not limited to preventing buffer overflows. C/C++ I/O functions such as scanf and printf can be replaced with their safe alternatives such as snscanf and snprintf, or _snprintf_s from SafeCRT library [127], or StringCchPrintf, StringCchPrintfEx, StringCchVPrintf, StringCchVPrintfEx from StrSafe library [138]; the replacement prevents format string attacks. Similarly, SQL injection attacks can be prevented by using SQL PreparedStatement [202] as a replacement for all instances of string concatenation based SQL queries in Java programs. Yet another instance of a safe function to prevent SQL injection is Magicquotes or addslashes functions in PHP that automatically escape quotes. A time-of-check-to-time-of-use (TOCTTOU) race condition error can be removed by replacing unsafe tmpfile and mktemp functions in C/C++ programs with safe mkstemp function (Unix) or tmpfile_s function (Windows).

Example

We have written a Perl script to search and replace strcpy and strcat functions and applied it on open source C programs.

Details of the script. The proof-of-concept Perl implementation transforms three types of files.
1. Source Code. It replaces `strcpy` and `strcat` functions with `g_strlcpy` and `g_strlcat` functions from the `glib 2.0` library. It uses two C functions, `sizeof` and `malloc_usable_size`, to calculate the size of stack buffers and heap buffers correspondingly. Our code identifies the source and destination parameters by pattern matching. Two patterns are illustrated in Table 7.3.

<table>
<thead>
<tr>
<th>Patt. Name</th>
<th>Parameters</th>
<th>Search Pattern</th>
<th>Replacement Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Function with two variables</td>
<td>Two parameters each of which are variables</td>
<td><code>strcpy (dst, src);</code></td>
<td><code>g_strlcpy (dst, src, sizeof(dst));</code></td>
</tr>
<tr>
<td>2. Function with pointer arithmetic</td>
<td>Any of the parameters have pointer arithmetic with integer</td>
<td><code>strcpy (*dst + index, src);</code></td>
<td><code>g_strlcpy (*dst + index, src, malloc_usable_size(*dst) - index);</code></td>
</tr>
</tbody>
</table>

Table 7.3: Patterns for `strcat` and `strcpy` functions

In some cases, it is impossible to determine the size of the destination parameter with simple pattern matching. Our proof-of-concept tool does not make any changes; instead, it points these cases out so that a developer can manually change the functions. However, it is possible to build a more comprehensive tool that covers these corner cases; the section on how to build tools illustrates these cases and describes how a commercial quality tool can be built.

2. Makefiles. The modified program has to compile correctly. The quickest fix is to include the library during link time. The script modifies `Makefiles` by looking for the pattern `gcc` as a single word and replacing it with `gcc `pkg-config --libs glib-2.0`. `.

3. Configuration Files. It transforms `config.status` file following the same pattern used for `Makefiles`.

Changes in the C code. The example section of Partitioning transformation lists a toy C program that goes through partitioning. The Safe Library Replacement transformation script was applied to the original program. The script replaces `strcpy` functions in lines 10, 13 and 41, and `strcat` functions in lines 14, 17, 36 and 42. Lines 35 to 42 are listed with the changes highlighted.

```c
34 ...
35 while(fgets(line, 80, fp) != NULL)
36     g_strlcat (fbuf, line, sizeof(fbuf));
37 fclose(fp);
38 pid = getpid(); // Get process id
39 sprintf(pidstr[0], "%d", pid);
40 g_strlcpy (filepath, "temp/", sizeof(filepath));
41 g_strlcat (filepath, pidstr, sizeof(filepath));
```

101
We have used the Perl script on two open source C programs: a pdf/ps file viewer (gv) and a zip library (zziplib). These programs have recent buffer overflow exploits [29] [30], and the exploit codes are available.

We examined version 3.6.2 of gv which has 46 C files with 27000 lines of code. There are 49 strcpy and 58 strcat instances in the source code. Our tool replaces 88 of these 107 instances. Patterns used in our tool are not sophisticated enough to replace the two cases. In the remaining 17 cases, 9 for strcpy and 8 for strcat, our proof-of-concept tool can not determine the size of the destination buffer. We manually changed the functions in these cases. The tool also changes 4 lines in the configuration file.

zziplib is a compression library. We used our tool on version 0.13.47 which has 7346 lines of code in 33 C files. The tool changes all 5 strcpy and strcat instances as well as 15 places in the Makefile.

In both cases, the resultant programs do not have buffer overflow vulnerability. They compile correctly, and show the same behavior. They pass all the test cases when we run the make test command.

Our proof-of-concept tool misses 19 out of 107 instances of strcpy and strcat when applied to gv version 3.6.2. In two cases, it fails to identify the source and destination parameters properly with string matching. These two cases were,

```
... strcpy((cwild=malloc(strlen(wild)+1)),wild); // FileSel.c: Line 672
...
```

and

```
... strcat(msg, count == 0 ? ", expecting `" : " or `"); // Aaa_bison.c: Line 1050
...
```

Writing more complex patterns improves the coverage of the tool. A more traditional design choice is to use abstract syntax trees to identify the parameters instead of relying on pattern matching.

In the remaining 17 cases, our tool can not determine the size of the destination buffer. This happens when the destination parameter is a pointer passed to a function. These cases can be handled by using safe library functions that rely on a new data type. A Safe Type Replacement (section 7.4.4) transformation handles these cases.
Another example of Safe Library Replacement transformation. Here, we will describe a Safe Library Replacement transformation on a C program that replaces unsafe arithmetic functions to prevent integer overflow errors. Several safe integer library options are available, but none of them are for ANSI C programs. The implementations of other libraries can be ported and rewritten for our tool’s operating environment. For example, the IntegerLib library constrains integer operations in the following way,

```c
unsigned short int total;
total = strlen(argv[1]) + strlen(argv[2]) + 1;
char *buff = (char *) malloc(total);
```

can be modified to,

```c
unsigned int total;
if (UAdd(strlen(argv[1]), 1, &total) && UAdd(total, strlen(argv[2]), &total)) {
    char *buff = (char *)malloc(total);
} else
    exit();
```

The safe integer library provides a collection of utility functions that can assist software developers in writing C programs that are free from common integer problems such as integer overflow, integer truncation, and sign errors. Functions have been provided for all integer operations subject to overflow (addition, subtraction, multiplication, division, unary negation, etc.) for int, long, long long, and size_t integers.

Towards a Safe Library Replacement Tool

Our proof-of-concept tool performs string matching on source code. A commercial quality tool needs to perform more robust static and dynamic analysis, perhaps a special purpose transformation language such as TXL [51], Stratego [211] and CIL [148]. An approach in this direction is the Gemini tool [56]. Gemini, written in TXL, applies a different transformation than ours. Instead of replacing functions, it transform all stack-allocated buffers in a C program to heap-allocated buffers, because exploiting a heap overflow is more difficult. Gemini does not completely remove the vulnerability. Nevertheless, Gemini’s use of special purpose program transformation languages is the right step towards designing commercial quality transformation tools.
Usually there are many safe alternatives for each unsafe function. Each safe function in Table 7.2 uses the same source and destination buffer parameters of the unsafe function; a size parameter is added to the original parameters in most cases. Another approach is to use functions with parameters of a new type [147, 93] (see the Safe Type Replacement transformation in section 7.4.4).

There are safe libraries available for preventing integer overflow vulnerabilities. Using a safe C++ class (such as SafeInt, IntSafe) or arbitrary precision arithmetic packages (such as GMP [78], CLN [46]) can relax integer security problems.

Another approach, that also replaces library, changes environment variables to dynamically load the libsafe library. Once preloaded by a vulnerable process, the library intercepts all C library functions, and allows a function to execute only if its arguments respect their bounds [13]. The mechanism of such a Safe Library Replacement transformation is more similar to a chroot Jail transformation (section 7.5.1) since both changes the environment of a process.

### 7.4.4 Safe Type Replacement Transformation

You have a program that uses a function that may cause data injection attacks if it receives an insufficiently validated input. A safe alternative for the unsafe library function is available, but it uses a different data type.

*Replace datatype of program variables so that a safe library function can be introduced in place of unsafe functions.*

**Motivation**

In many domains, programmers are used to replacing old libraries in a program with new libraries, e.g. replacing one library with another to use a different database, replacing libraries for changing mobile standards etc. It will be easier to automate the process (a Safe Library Replacement transformation) if the new library has similar signatures.

In many cases, the transformation is not straightforward. The new library may require different datatypes. qmail prevents buffer overflow by replacing unsafe string functions with a new library. The new library uses a separate data structure [93] that keeps length and allocated memory information of string buffers. Before introducing the new library, the datatypes have to be changed to enable the transformation.
Preconditions

A program with the following characteristics benefits from a library replacement transformation

1. The program uses unsafe functions for which safe alternatives are available.
2. The safe library alternative uses a different datatype.

Mechanism

For each unsafe function, a developer specifies the alternative safe function and the safe library.

The program transformation finds all functions that need to be replaced. It replaces the datatype of function parameters so that the safe function can be introduced; then it replaces unsafe functions with suitable alternatives.

Example

We will describe two examples where the transformation is applied. First, it is applied to prevent C buffer overflow attacks. Second, it is applied on a Java program to prevent SQL injection attack.

Preventing buffer overflow in a C program. We have developed a proof-of-concept tool for replacing strcpy and strcat functions. We have described how the tool has been applied on various programs in the Example section of Safe Library Replacement (section 7.4.3) transformation. When the tool has been applied to gv, it can not replace 17 instances of unsafe functions. This happens when the destination parameter is a pointer passed to a function. Here is a toy example that describes this situation.

```c
#include <stdio.h>
#include <string.h>
#include <malloc.h>

void foo(char* dest) {
    strcat (dest, "Hello World");
    printf("%s", dest);
}

void main (int argc, char *argv[]) {
    char a[7]="";
    char *b;
    b = (char *) malloc (10);
    foo(a);
    foo(b);
    free(b);
}
```
The dest parameter in line 6 can point to a variable allocated in stack (line 14) or heap (line 15). A commercial quality tool can apply program slicing techniques to identify the type. In the example, a backward slice indicates that the variable can be allocated in stack or heap. To resolve this issue, the foo function can be inlined replacing the function calls. But in more complex cases, alternative safe functions that use separate data types to keep memory information can be used. An example is the safestr_* functions from Safestr library [135]. The transformed program is listed here,

```c
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <safestr.h>

void foo(safestr_t dest) {
    safestr_t str3;
    str3 = SAFESTR_TEMP("Hello World");
    safestr_concatenate (&dest, str3, 0);
    printf ("%s", (char *) dest);
}

void main (int argc, char * argv[]) {
    char a[8] = "";
    char *b;
    safestr_t str1;
    safestr_t str2;
    b = (char *) malloc (10);
    str1 = safestr_create(a, SAFESTR_ASSET_PERMANENT);
    str2 = safestr_create(b, SAFESTR_ASSET_PERMANENT);
    foo(str1);
    foo(str2);
    free(b);
}
```

In the example, new datatypes are created (lines 8, 19 and 20) for using the safestr_concatenate function in line 9.

**Preventing SQL injection in a Java program by introducing PreparedStatement.** In Java programs with SQL injection attacks, all instances of query string formulation can be replaced with pre-compiled SQL statements represented by Java PreparedStatement. Consider the Java code in the Example section of the Decorated Filter transformation (section 7.4.1).

A developer can apply a Safe Type Replacement transformation on this program. The transformation introduces Java PreparedStatement replacing lines 11-12, where the SQL query was formed by string concatenation. Here is the modified code snippet,
Towards a Safe Type Replacement Tool

A Safe Type Replacement tool requires in-depth program analysis. These tools are more likely to be targeting a particular replacement. For example, a tool can target C programs that use `char *` datatype and introduce `safestr_t` data structure. On the contrary, a Safe Library Replacement tool for C programs can replace multiple unsafe functions.

In the C buffer overflow examples, the replacement types typically keep more information about the buffer. The `stralloc` data structure has been used in qmail [93]. This data structure keeps the length information of the string buffer. The `len` field keeps the length of the buffer in bytes and the `a` filed keeps the count of allocated bytes in the string.

```c
typedef struct stralloc {
    char *s;
    unsigned int len;
    unsigned int a;
} stralloc;
```

All string-manipulation functions check if the input data fits the buffer before attempting to update the buffer. Another mail transfer agent, `sendmail X`, has a data structure named `sm_str_S`, that keeps length information [93].

Pre-compiled SQL statements in Java has been used in other languages too. Previously, PHP programs use the mysql library to query mysql database. The querystrings are typically generated using string concatenation with insufficient or zero validation of user inputs. mysqli library has been introduced to replace mysql library to prevent SQL injection in PHP programs. The library uses modified datatype and pre-compiled SQL statements.
7.4.5 Randomization Transformation

All applications using some off-the-shelf components fall prey to an exploit in the component because they use the component in the same manner. You want to protect your system from these common exploits.

Make your system unique so that it does not fall prey to a common vulnerability.

Motivation

Software applications are distributed by code cloning; all applications look the same and act the same. Homogeneity is beneficial for software maintenance, but diversity in software makes software more secure against an attack. It is similar to biological systems; where every entity is different, and hence the same virus does not have the same impact on every entity. Opportunities of diversity may create some strains that are not vulnerable to an exploit. It also makes the task harder for an attacker to target a system and come up with an exploit, because he (or she) does not have the knowledge of randomized system internals.

However, the opportunities of automatically varying a program is limited.

Preconditions

A program with the following characteristics benefits from a randomization transformation

1. The program or platform supports limited number of randomization opportunities.
2. The program often comes under known injection attacks, such as SQL injection or buffer overflow attacks.
3. An attacker cannot know the secrets that are introduced by randomization of program artifacts.

Mechanism

A developer specifies an artifact of the program (e.g. a data variable, instruction set etc.) that he wants to make distinct.

The transformation randomizes the artifact in order to diversify the system so that it survives common vulnerabilities.
Example

The following two examples describe two randomization transformations to prevent buffer overflow and SQL injection attacks.

Preventing Buffer Overflow Attack. In order to come up with a buffer overflow attack vector an attacker has to have some intimate knowledge of the stack architecture. To keep an attacker guessing, a program can be randomized by modifying the stack allocation process. A Randomization program transformation applies to the assembly code representation of a C program. It transforms assembly code by adding a random number of functions that are side effects free [75], e.g. NOP instructions, XOR of same values, INC/DEC pair etc. The resulting software changes the expected address of the targeted instruction, instructions that the attacker attempts to jump to by overflowing the buffer.

Preventing SQL Injection Attack. A program transformation can introduce a translator proxy between a web server and a database. It transforms a program so that all the SQL keywords generated by the system are augmented with a common random postfix. An attacker, who passes an attack vector containing SQL keywords, will fail to guess the random postfix. When the proxy finally generates the SQL statement, the postfixes are stripped. Thus the attackers injection is identified. A policy to not execute a tainted SQL statement prevents SQL injection errors.

This technique is implemented by the SQLRand [25] system. In SQLRand, an example query are created this way,

"select123 * from123 table1 where123 name='"+user_input+"'"

An attacker passing "'and'1'='1" for user input gets the following query

select123 * from123 table1 where123 name='" and '1'='1'

When a translator proxy deciphers it will expect and123 instead of and; it determines that the query is invalid.
Towards a Randomization Tool

Tools already exist that modify a program to prevent buffer overflow [75] and SQL injection attacks [25]. Further opportunities of randomization should be explored. Typically, a Randomization transformation targets the function layout and variable layout of a program. By varying them, the transformation creates customized versions.

Another way to prevent buffer overflow attack is to allocate a random marker to denote data endpoint in stack and heap [54, 53]. The program dynamically examines the markers for corruption. A corrupt marker indicates an overflow. An attacker cannot guess this random marker.

Other than inserting statements and markers to prevent buffer overflow attacks, a program transformation can randomly reorder basic blocks in a program. This can be used to prevent some virus attacks. A fine grained example of reordering is modifying code execution order within a basic block [153].

Several other techniques thwart various types of injection attacks and malware propagation [69]. Object files such as `crt0.o` that are executed before user code can be altered. Also changing the names and argument sequence of dynamic libraries can create customized binaries for each machine. Another way to create customized binaries is to vary the magic number of executables. Yet another opportunity is to vary the name of important system files, e.g. the password file.

The transformation is motivated from research on instruction set randomization [115]. By randomizing underlying system’s instructions, ‘foreign’ code injected by an attacker will fail to execute correctly regardless of the type of injection. This is not only limited to SQL injection, but also applies to code injection attacks. Randomizing an arbitrary instruction set involved three components: the randomizing element, the execution environment and the loader. Following the same approach, RISE [14] uses an emulator attached to processes to create unique binary code randomization for each process. The obfuscation is done at the machine emulator level. A translator trivially transforms randomized code to machine executable code, but an attacker’s injection becomes useless. Perhaps not directly related, but executable code obfuscation has also been studied to prevent static disassembly [125] so that attackers do not learn by reverse engineering code. Another important recent work is continuing on automatically locating and repairing bugs in production software with genetic programming [216].

\[^5\text{crt0.o calls the main function.}\]
Randomization has two distinct characteristics. First, the transformation is a mixture of various tricks, e.g. random insertion, reordering, using a random marker etc. Other program transformations have one specific mechanism. Second, most of the randomization mechanisms apply on binaries.

7.5 Protecting Resources

7.5.1 chroot Jail Transformation

You want to run a process in a constrained environment.

Make the files in the filesystem inaccessible to a process.

Motivation

In the first week of September 2008, Bugtraq listed 14 unsecure file creation vulnerabilities out of a total of 154 vulnerabilities; a chroot jail can be used to fix the problem in 13 of these cases.

Postfix has two processes to handle its mail queue: a writer process and a reader process. Both processes run with lower privilege. But, an attacker controlling a writer process may read, write or execute arbitrary files not supposed to be accessed by the process. Postfix prevents this by running its writer process inside a chroot jail.
Partitioning does not prevent an attacker from accessing files; a process should also run inside a sandbox. One technique is to run a process inside a chroot jail [72]. A chroot operation changes the root directory of a process. Since a process cannot access files that are above its new root, it cannot read, write or execute those files.

A chroot operation is implemented as a system call or as a wrapper program. Both implementations take a directory name as parameter, and make it the root directory for a process. A chroot(2) system call in a program’s source code, when executed, constrains a process inside a jail. On the other hand, a wrapper program is invoked by a shell script. A shell process creates a jail, calls a target program through a system(3) call and shells out to it.

Creating a jail environment automatically is an unconventional program transformation, as it makes very few changes to source code or binary. Instead, it transforms the environment of a process by analyzing its source code and binaries and finding its dependencies (e.g. libraries, resource files etc). The only conventional part is to transform source code to include a chroot system call, or create a wrapper shell script to shell out to a binary.

**Preconditions**

A program with the following characteristics benefits from a chroot jail transformation.

1. The process runs in Unix environment.
2. Processes communicate using shared files.
3. At the chroot call insertion point, the program does not have open file descriptors to directories outside the jail.
4. The program can run as a user with limited privilege.
5. The program depends on a small number of files and libraries. It is possible to set up an elaborate jail environment [43], but specifying permission level for a large number of files is problematic.
6. Path to all data files can be resolved statically.
Mechanism

A developer specifies the program that is to be jailed, the programs that communicate with the jailed program using shared file, location of the chroot directory, location in program where the chroot call is inserted, and the privilege level of the jailed program and the communicating programs.

The program transformation creates a jail environment and run a program inside it. The steps are described next.

Creating a Jail Environment. Create a minimal jail environment following these steps.

1. Replicate a program’s directory hierarchy, and copy the jailed program in the appropriate directory.

   The entire directory hierarchy may not be needed for the program. This step is necessary only when an operating system creates a chroot jail for every untrusted user, e.g. the jail in FreeBSD [114]. Creating a configuration mini-language to differentiate between this special case with others is infeasible because of the additional complexity. A few mkdir calls do not impact the overall performance of the program transformation.

2. Copy shared libraries. Prepare a list of shared libraries by querying a binary for shared libraries, e.g. use the ldd(1) command for ELF type binaries. Alternatively, parse makefiles and make configuration files to prepare a list of shared libraries. Copy the library files in the appropriate directory hierarchy.

   Do not use ldd on untrusted programs. ldd sets a special environment variable before executing a program. An untrusted program may set the environment variable erroneously.

   ldd generates a segmentation fault with cross compilation. A modified version of ldd is used for these special cases. For example, uClibc is a replacement of the glibc library for embedded systems. ldd generates a segmentation fault when it runs on an application dependent on uClibc. Instead, uClibc provides its own implementation of ldd; the modified ldd does not generate a segmentation fault.

3. Copy resource files and configuration files. Search for fopen(3) system calls in source code for a list of resource and configuration files and copy them. Trace binaries with strace(1) command.

6LD_TRACE_LOADED_OBJECTS for ELF objects
The `strace` lists system calls made by the process. First, create a chroot jail and trace a program running inside the jail. Every time the program fails to open a file, it will log an error message in `strace` output. Parse these error messages, copy the listed files, and run `strace` again. Resolving some dependencies may open up more dependencies; repeat the whole process until there are no errors.

4. Copy dynamically loaded libraries. Search source code for `dlopen(3)` calls to make a list of dynamic libraries. Otherwise, run `strace` on binaries similar to step 3.

5. Change file access permissions so that a jailed program does not run as root [72]. Start a program as root and then lower its privilege level.

Change permissions of files inside a jail. The simplest way is to start with everything owned by root and gradually open privilege. A less privileged user should have read, write and execute permission on directories, read and/or write permission on files, and execute permission on binaries. For complex permission distributions, create a configuration mini-language.

**Running a Program inside a Jail**. A chroot jail can be created by a system call or a wrapper program.

For a system call, insert a `chroot` call with the name of a jail directory as parameter. After that, insert two `chdir(3)` calls: first a `chdir("/")` call to change to the new root followed by another `chdir` call to change the directory to the working directory of the program. Finally, insert a `setuid(2)` and `setgid(2)` call with a non-root user id to lower the privilege level. Close all open file handles and avoid the use of symbolic links at the point of the `chroot` and the `setuid` call. Finally change the path of the shared file(s) to absolute path in all the processes that communicate with the jailed process.

A wrapper program calls a jailed program and transfers control to it. Inside a wrapper, make a `chroot` call, lower the privilege level and shell out to the jailed program.

**Example**

We have written a general-purpose Perl script to automate the first phase. The script takes an arbitrary program and creates a jail environment for the program. All of these steps involve running Unix commands and parsing their output. The proof-of-concept tool does not change file access permissions; it is done manually. Also, we manually inserted `chroot` (and other supporting) calls in the program, but a tool can be built to automatically do this.
The example section of the Partitioning transformation (section 6.1.1) describes how a C program was partitioned between a reader program and a writer program. Suppose a developer wants to run the writer program from that step in a chroot jail. The specification for the program transformation should include:

1. The location of the chroot directory (suppose it is /chroot/jail).
2. The program to be jailed (the writer program is in /home/writer), and the programs that communicate with it (the reader program is in /home/reader).
3. The line number of the program where the chroot system call has to be inserted (line 13).
4. The privilege level of the writer (uid 1001 and gid 101) and the reader process (uid 1002 and gid 101).

Creating a Jail Environment. First the tool creates bin, lib, dev and tmp in /chroot/jail. Then it copies the writer program in /chroot/jail/home/writer.

Then the tool runs ldd to find shared libraries, and repeatedly runs strace to find resource files, configuration files and dynamic library files. It copies the files in /chroot/jail.

The writer process creates new files in temp directory (/chroot/jail/home/writer/temp directory in this example). To set permissions, we make the writer process the owner of this directory. Then we grant read, write and execute permission to other users.

Running a Program inside the Jail A chroot call is inserted in line 15 of the writer program, followed by chdir, setuid and setgid calls (the changes are highlighted).

```c
... fclose(fp);

chroot("/chroot/jail"); // chroot call
chdir ("/"); // Change directory
chdir ("/home/writer"); // Change directory
setuid(1001); // Set user id
setgid(101); // Set group id
pid = getpid(); // Get process id
...
```

Finally we change the path of the shared file in the reader process (10 and 13).
Towards a chroot Jail Tool

There are many tools that automate different parts of chroot jail setup process. Jailkit [187] and Jailtool [205] limit user accounts to a specific directory and to specific commands. FreeBSD’s jail facility [114] partitions the operating system’s running environment into a management environment, and optional jail environments. FreeBSD’s build environment sets the jails up. Each of these only solves a part of the problem. A chroot Jail program transformation relieves a developer from many manual steps involved in jail creation.

Our proof-of-concept tool automatically creates the jail environment for a program. The second phase involves adding new source code but it follows a strict template. It is not difficult to build such a tool. This program transformation requires a lot of specification, especially if privilege distribution is complex. Creating an efficient domain-driven specification language and implementing an efficient parser will further improve the tool.

7.5.2 Unique Location for Each Write Request Transformation

Unreliable locking mechanism can lead to corrupt files when multiple writer processes are trying to access a shared file. You want to protect data in shared files.

Create unique files for each write request so that writer processes are never accessing the same file at the same time.

Motivation

When multiple processes try to write some data concurrently to the same resource, there is need for some locking mechanism. But the locking mechanism is not always atomic and a crash may leave a trace in the resource. It can be useful to create separate entry for each information write request - if the separate entries are within a manageable limit.

```c
...  
g_strlcpy(dirpath, "/chroot/jail/home/writer/temp", sizeof (dirpath));
dirptr = opendir(dirpath);    // Open directory  
while (direntry = readdir(dirptr)){  
    g_strlcpy (filepath, "/chroot/jail/home/writer/temp", sizeof(filepath));
...
```
Consider a mail queue implemented as one shared file. Multiple processes accept email and then attempt to access the shared file and write mail to the queue. Ideally, the locking mechanism and writing to the shared file should be atomic, but in practice the locking mechanisms do not work portably and reliably. An example is the ‘dotlocking’ mechanism Sun’s Network File System (NFS). NFS exacerbates all the problems. Some NFS implementations even don’t provide any reliable locking mechanism.

Early versions of sendmail have a mail queue that is a single file. It uses OS-provided locking mechanism for file access. But a machine may crash while it is delivering a message. This means that the message will be silently truncated in the shared file and corrupt it.

In order to prevent this corruption, all file writing should be atomic and orthogonal.

**Preconditions**

A program with the following characteristics benefits from this transformation

1. The program has processes communicating with shared files. Multiple writer processes access a shared file by locking the file, and reader processes periodically poll the shared file for newly written data.

2. The locking mechanism does not work efficiently for handling concurrent calls.

3. Reader and writer processes are not coupled with a particular file or file name. They can communicate with any arbitrary files.

**Mechanism**

A developer specifies the sections of a program that accesses a shared file in the reader and writer process and new file creation policy. He or she also specifies information about the shared file.

The transformation modifies the write request so that a new file is created for each write request. In the writer process, it modifies the file open function with a new file creation function that uses a new file name in each call. The new file name generator is a separate component that generates a unique file name on every request. In the reader process, the file open function is replaced with a read directory function, to poll the directory where the new files are created.
This program transformation replaces the unreliable file locking attempts with separate file creation calls that never contend. This separation improves reliability. It also ensures that the traces of a crash do not affect any central file. Locking does happen under the hood, since the file creation task needs to put a lock on the directory. However, Network File System (NFS) ensures that this locking and file creation under a directory is an atomic process. This mechanism is more reliable than placing a lock on a file.

Example

The first example in the Partitioning transformation (section 6.1.1) describes a reader and writer program communicating with one shared file. The file in the writer process is opened in line 20 of the original listing.

```c

fp = fopen (filepath, "w"); // Create file for writing
fwrite (fbuf, sizeof(fbuf[0]), strlen(fbuf), fp);
// Write to file

```

A developer selects the fopen call and specifies that it be replaced with a unique file. The program transformation replaces the filepath parameter with a parameter that is generated at runtime. It calls a unique file name generator component.

In the example, there is only one writer process. If there are multiple processes then some locking mechanism is also present in the code. These are removed in the transformed program.

The strategy to generate the unique file name can also be specified to the program transformation. For example, the following lists a unique file creation algorithm that concatenates the current time, process id and hostname to create a file name. It creates the new file inside the temp directory. [The code is an actual excerpt from qmail.]

```c

//Main loop for unique file creation
for (loop = 0;;++loop)
{
    time = now();
    s = filepath; // s is a pointer to the filepath parameter
    s += fmt_str(s, "temp/");
    s += fmt_ulong(s, time); *s++ = ‘.’;
    s += fmt_ulong(s, pid); *s++ = ’.’;
    s += fmt_strn(s, host, sizeof(host)); *s++ = 0;
    ...
```
The reader process is already polling the temp directory for new files. So no change is needed in the reader process.

Towards a Unique Location for Each Write Request Tool

Other than the unique file name generator component, the other parts of the transformation are removing and replacing lines of code, and calling the file name generator component.

It is possible to use a boilerplate file name generator, or the transformation can take a specification from a developer and create a file name generator accordingly. The example shows how unique filenames can be created by appending time, pid and host. Various schemes of concatenation of any number of following strings have been employed by modern delivery identifiers to absolutely guarantee the uniqueness. The following lists some parameters available in the Unix environment.

- \#n, where n is (in hexadecimal) the output of the operating system’s `unix_seqeucnumber()` system call, which returns a number that increases by 1 every time it is called, starting from 0 after reboot.

- Xn, where n is (in hex) the output of the operating system’s `unix_bootnumber()` system call, which reports the number of times that the system has been booted. Together with #, this guarantees uniqueness; unfortunately, most operating systems don’t support `unix_seqeucnumber()` and `unix_bootnumber()`.
• Rn, where n is (in hex) the output of the operating system’s `unix_cryptorandomnumber()` system call, or an equivalent source such as `/dev/urandom`. Unfortunately, some operating systems don’t include cryptographic random number generators.

• In, where n is (in hexadecimal) the UNIX inode number of this file. Unfortunately, inode numbers aren’t always available through NFS.

• Mn, where n is (in decimal) the microsecond counter from the same `gettimeofday()` used for the left part of the unique name.

• Pn, where n is (in decimal) the process ID.

7.6 Testing Systems for Robustness against Injection Attacks

7.6.1 Fuzzing Transformation

You want to test whether a system can survive attackers’ attempt to crash it.

Add a fuzzing component that randomly generates data and runs the program with that data.

Motivation

The fundamental principle behind all injection attacks is to run an application with an input that it cannot execute. Ideally, an application should only process inputs that does not produce an erroneous state, i.e. inputs that are in its comfort zone [171]. But applications are designed with a relaxed input domain, and they process inputs outside the comfort zone. Part of the reason behind the relaxed input domain is that the developers fail to determine which inputs are safe, and which are unsafe. Testing an application with a few “chosen” inputs depends on the “choice” of a developer.

Testing a system with invalid, unexpected, or random data is not an alternative to exhaustive testing and formal methods, but it complements the works of a tester. Fuzz testing can only be regarded as a bug-finding tool: in many cases passing a fuzz test may only demonstrate that a piece of software handles exceptions without crashing, rather than behaving correctly. Fuzzing does not eliminate the need for exhaustive testing.
Preconditions

A program with the following characteristics benefits from a fuzzing transformation

1. The program can only be tested as a black box.

2. The program has to be tested for crashes, assertion failures, and memory leaks. Fuzzing is a useful technique for a C/C++ application where the impact of a memory safety violation results in a security vulnerability.

3. The program has to be tested for the correctness of its error-handling routines.

Mechanism

A developer specifies the program insertion points where fuzzing tests are to be applied. Typically, fuzzing is applied on code that processes input that is received across trust boundaries. A developer’s specification includes these trust boundaries, e.g. files received from the Internet, network sockets, pipes, RPC interfaces, driver IOCTls, ActiveX objects, configuration files that can be touched and edited by external users, database entries made by external users, message queues, shared memory etc. A developer also specifies the policies that are used to generate inputs.

The transformation automatically adds a component that generates random inputs based on user specification. The component feeds the inputs to a system and reports results.

Example

The first example in the Partitioning transformation (section 6.1.1) describes a reader and writer program communicating with one shared file. The reader file processes inputs (prints it to stdout) read from the shared file which are generated by user inputs from a writer program. The inputs from the writer program are untrusted.

We will describe how a developer can apply a Fuzzing transformation to test the reader file. At first, a developer chooses the input processing section from the reader file.

Part of the reader file is listed here. In line 16, the program reads untrusted input from the shared file and prints it in line 18. The program is vulnerable from a buffer overflow attack in line 17 and a format string attack in line 18.
... while(fgets(line, 80, fp) != NULL)
    strcat(buf, line); // Read file content
    printf("%s\n", buf); // Print file content
...

The developer selects the variable line and specifies that a fuzz testing component is used to generate random inputs. The program transformation introduces an input generation component and uses its inputs to run and test the program. The developer may specify policies for generating inputs. For example, he may specify that the system be tested for buffer overflow and format string attacks, and the input generation component will produce inputs that are likely to reveal those errors. The developer can also provide more fine grained specification, e.g. length of inputs, special characters that can be introduced in inputs etc. A specification can also include a set of valid inputs, that can be used by a mutation fuzzing engine.

Both the attack can be mitigated by a Safe Library Replacement transformation that replaces unsafe functions, or a Decorated Filter transformation that validates inputs before they are used in functions. A developer will find the injection vulnerabilities applying the Fuzzing transformation, and then will possibly apply both Safe Library Replacement and Decorated Filter transformations to mitigate the vulnerabilities. Additionally, the inputs that crash/exploit the system in the testing can be used by the developer to better ascertain the “comfort zone”; he (or she) may also use them to create smarter filters in a Decorated Filter transformation.

Towards a Fuzzing Tool

Several commercial and open source fuzzing tools can be used by a Fuzzing transformation. Fuzzing tools typically fall into three categories: fuzzing frameworks, general purpose tools and special purpose tools. A Fuzzing transformation tool can use a fuzzing framework to generate a new fuzzing tool based on user specification, or it can parse user specification and use it to regulate general purpose or special purpose fuzzing tools.

The transformation can use several fuzz testing frameworks, e.g., antiparser [8], Autodafé [129], Peach [157], SPIKE [2], sulley [198], etc. Among general purpose tools, beSTORM [16], Codenomicon [48], GPF [80], MU-4000 [195], ProxyFuzz [166], etc are prominent. Prominent special purpose tools include ftpfuzz [73] for FTP server testing, PROTOS [164] for SNMP protocol testing, etc.
7.7 Organizing the Transformations

Program transformations described in this chapter are organized using two criteria: their impact on codebase and the type of transformation. Table 7.4 describes how they are organized.

7.7.1 Impact on Codebase

**Small change.** Program transformations in this chapter make small changes to the existing codebase, specifically, the code where input data is received. Some of them introduce large components (e.g. a Fuzzing transformation introduces an input generation component, a Unique Entry for Each Write Request transformation introduces a unique file name generator component), but the change in the original source code is small and includes delegations of tasks to the added component.

Transformations that make small, frequent changes are the most suitable candidates for automation. A Safe Library Replacement transformation can be fully automatic. The transformation replaces all instances of an unsafe function with a safe function. Unsafe functions in C exists in string operations (strcpy, strcat), formated I/O (printf), memory management (memcpy), temp file creation (tmpfile), etc. Replacing these functions with safe alternatives (e.g. g_strlcpy, g_strlcat, vsprintf, memcpy_s, mkstemp etc.) prevent buffer overflow errors, format string attacks and TOCTTOU race condition attacks.

Other transformations are less frequent; they only make changes in a few places of an application. Transformations that add input validation components at system perimeter fall into this category. chroot Jail transformation, also in this category, makes changes to a few application components.

<table>
<thead>
<tr>
<th>Add</th>
<th>Small, infrequent change</th>
<th>Small, frequent change</th>
<th>Large change</th>
<th>Change beyond boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add Perimeter Filter</td>
<td>Message Intercepting Gateway</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Limit</td>
<td>chroot Jail</td>
<td>Explicit Type Enforcement</td>
<td>Safe Library Replacement</td>
<td>Safe Type Replacement</td>
</tr>
<tr>
<td>Mutate data</td>
<td>Decorated Filter</td>
<td>Fuzzing</td>
<td>Randomization</td>
<td></td>
</tr>
<tr>
<td>Distribute</td>
<td>Unique Location for Each Write Request</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.4: Security-oriented Program Transformations for Unvalidated Input
7.7.2 Type of Transformation

**Add component.** Since the changes have a small footprint, developers have the option of applying a transformation manually or using tools. For example, a developer can *Add Perimeter Filter* component at the system entry point manually or automatically. This transformation should be applied to a system that has few entry points, i.e., few places where input data enters the system. (To ensure that a system has few entry points, it is beneficial to first apply the *Single Access Point* transformation described in section 6.2.1.) The code change involves adding a filter class or component and calling it from each entry point. A tool can automatically add the component and the call to it given a developer has identified the system entry points. The transformation made by a tool does not require deep understanding of program behavior that is required to identify the entry points.

**Limit.** A somewhat different transformation is the *chroot Jail* transformation, which limits a Unix process from writing beyond a directory hierarchy. This is an atypical transformation, because it requires very few changes in the source code or binary. Instead, the program transformation creates a jail environment for a process. The transformation analyzes a program and finds its dependencies on static and dynamic libraries, file descriptors, etc. The tool can also properly add the *chroot* calls in the source code so that file descriptors are closed across the call, the privilege of the calling process is lowered, and the shared resources have proper permissions.

**Mutate data.** Program transformations that mutate data require developers to specify the policies that modify (or mutate) input data. A *Decorated Filter* transformation adds input validation policies, specified by developers, to decorate [74] an input variable. These policies are injection attack specific; thus they can be easily extended as new injection attacks emerge. A tool for this transformation is very simple – it automates the *Moving Embellishment to Decorator* [71] refactoring, which is a purely structural change. A tool does not need deep understanding of program behavior since it does not identify by itself where to apply the transformations or which input validation policies to apply. These are specified by the developers.

**Distribute.** Some program transformations break a single point of failure into multiple entities. A *Unique Location for Each Write Request* transformation prevents multiple processes from writing to one file by creating a unique file for each write request. This transformation creates a temporary file for each write...
task, writes data to it, and saves the file with a unique name. It also modifies the file read mechanism for the data consumers.

7.8 Composing the Transformations

Most of the transformations described here are orthogonal to each other; they can be applied in any order. For example, four program transformations, Decorated Filter, Explicit Type Enforcement, Safe Library Replacement and Fuzzing can be applied in any order. Decorated Filter, Explicit Type Enforcement and Safe Library Replacement transformations affect separate parts of the program: the first modifies input variables, the second modifies type declarations and the third modifies functions. Although Fuzzing can be applied any time to test an application, it is more efficient to apply it before and after applying the other two transformations to test the impact of transformations. Additionally, a developer can apply Fuzzing in tandem with Decorated Filter: each time a developer applies Fuzzing to determine better filters, then he (or she) applies the filters with Decorated Filter transformation, and checks the application with a Fuzzing transformation, possibly following this order in several iterations.

Safe Type Replacement transformation is a pre-cursor to the Safe Library Replacement transformation. It introduces new datatypes that are used by a new, safe library.

In many cases, Decorated Filter, Safe Library Replacement and Safe Type Replacement are used together to prevent an injection vulnerability. For example, a C developer can apply Safe Library Replacement function at first to remove unsafe library functions. A tool can safely replace a portion of all functions. The developer can then apply Safe Type Replacement transformation. This can still leave some corner cases. These corner cases can be checked by the developer, who can then write checks on inputs. He or she can use a Decorated Filter transformation to validate inputs before they are passed to the unsafe functions.

When a developer uses a Unique Location for Each Write Request, it is usually preceded by a Partitioning transformation (section 6.1.1) that partitions a process into several processes communicating with a shared file.

Finally, a Randomization transformation usually applies to binaries. It is applied last, after all the program transformations are applied on source code and a binary is produced.
7.9 Summary

This chapter describes 10 security-oriented program transformations. Applying them can protect software application from injection attacks, the most prominent attacks on software applications.
The goal of authentication is to verify that a user’s identity is what the user claims to be. The verified identity is used to determine the set of privileges belonging to a user. An attacker, who manages to establish a fake identity, can further damage a system with his (or her) improper set of privileges.

The goal of an attack on an authentication mechanism is to establish an improper identity, typically by attempting to guess a valid authentication token, e.g. a username password pair. The simplest prevention technique against brute force attacks is to limit the number of an attacker’s tryouts [118]. One would expect such a simple mechanism to be deployed in every software, but this is not the case.

Six program transformations are described in this chapter. One of them introduces an authentication enforcer component; another limits the guessing attempts. The remaining four program transformations add single sign on feature among multiple applications.

8.1 Broken Authentication Vulnerability

Broken authentication vulnerability is typically manifested as an access control vulnerability, since an attacker breaks the access control mechanism after establishing a false identity. We surveyed the Bugtraq vulnerability list [34] on several random weeks and found that on average 2% attacks are related to broken authentication. Another 5% on average are related to broken access control vulnerability; typically these are closely related to the authentication problems. This is not a high number. However, stopping the attackers at the system perimeter by properly authenticating them removes a lot of attackers from entering the system and launching other attacks. A proper authentication mechanism is therefore very important.

The focus of this chapter is on 3 authentication problems faced by developers.

1. Developers want to introduce an authentication component in an application. Developing an authentication component in any platform is a complicated task. Typically, developers use an authentication
framework suitable for their platform and compose classes of that framework. Since the frameworks do not enforce any regulation, often an authentication component is implemented in a non-standard way. Thus a program may be hard to maintain in the long run.

2. Developers want to prevent attackers from stealing users’ passwords. Authentication tokens, such as passwords, often come under brute force attacks. Three factors make this attack easy: 1) users choose passwords that are easy to guess, 2) users are duped by phishing attempts and reveal their passwords, and 3) applications do not protect their users from brute force attack attempts. While a developer cannot regulate user behavior, he (or she) may implement account locking mechanism to prevent brute force attacks. But this is not the case for many applications.

3. Developers want to eliminate too many passwords so that user’s have to remember few passwords. They typically develop single sign on feature in a simple and standardized way. Single sign on brings together the identity management functionality of diverse services. Currently, developers do not have any systematic method to implement various aspects of single sign on. The components, thus created, are often tightly coupled, and they often reveal internal information.

8.2 Solution Strategies

We describe 6 security-oriented program transformations to help developers in three contexts: 1) when a developer wants to introduce a new authentication component, 2) when a developer wants to prevent brute force attacks on authentication components, and 3) when a developer wants to add single sign on feature.

An Add Authentication Enforcer transformation introduces an authentication component by composing classes of a framework. It assumes that the developers want to use an existing framework for authentication, and it generates a default authentication component using classes in that framework. A developer specifies where the authentication information comes from and the authentication mechanism to be used, and the program transformation composes a default component. Automating this task ensures that the classes in an authentication framework are composed in a standardized way.

The mechanism to prevent brute force attacks is to put a limit on the number of authentication tryouts; an Add Account Lockout transformation adds this feature to an application.

The remaining four program transformations introduce the single sign on feature. These program trans-
formations do not prevent any particular vulnerability, rather these ensure that applications can use single sign on feature in a secure manner.

Most of the program transformations described in this chapter make large changes to an application. They are hard to automate. In fact, only part of the entire process can be automated. These program transformations require more developer interaction; they describe how developers can transform a program combining both manual and automated effort.

8.3 Adding Authentication Components at System Perimeter

8.3.1 Add Authentication Enforcer Transformation

You want to ensure that only a set of permitted users enter a system and each user has a separate identity.

*Introduce a component which authenticates a user and creates a subject to represent the user.*

Motivation

During authentication, a system verifies the identity of a requester using credentials. After successful authentication, a subject is created to represent the requester. The subject is populated with appropriate principals and credentials. The authentication mechanism can be different for different contexts, e.g. username-password based authentication, client certificate based authentication, or Kerberos based authentication. Irrespective of the mechanism, the authentication logic should be concentrated in a single component that can be created based on user specification.

Preconditions

A program with the following characteristics benefits from an *Add Authentication Enforcer* transformation

1. The program has a limited number of access points and policy enforcement points where an authentication component will be introduced. A developer may apply *Single Access Point* (section 6.2.1) and *Policy Enforcement Point* (section 6.2.2) to create policy enforcement points.

2. The developer wants to introduce authentication using an authentication framework.
Mechanism

A developer specifies the insertion point of an authentication component, source of input, authentication mechanism, and outcome of a successful or a failed authentication.

The transformation composes classes of a security framework (JAAS, .NET etc) to create a component with the specified authentication mechanism, and delegates authentication requests from the insertion point.

Example

Suppose, an imaginary developer Alice has a Java program that connects to a hostname and a port via a socket. Alice designed the program for the employees of her company; everybody was trusted, everybody had the same rights, hence there was no need for authenticating users. Now, Alice has to retrofit authentication, because her company is planning to let external users use the program.

Before applying the Add Authentication Enforcer transformation, Alice has applied the Single Access Point (section 6.2.1) and Policy Enforcement Point (section 6.2.2) transformations. Now she wants to introduce an authentication component which authenticates using username and password.

The Policy Enforcement Point transformation creates a PolicyEnforcementPoint class that delegates authentication request to an AuthenticationEnforcer class. This placeholder class is the starting point of the Add Authentication Enforcer transformation.

![Diagram](image_url)

Figure 8.1: Classes added by an Add Authentication Enforcer transformation

130
Alice replaces the AuthenticationEnforcer with a group of classes that perform username and password based authentication (see Figure 8.1). The Add Authentication Enforcer transformation composes the classes of the JAAS framework to provide a default authentication scheme. In our example, Alice specifies authentication type (username/password), source of user inputs, authentication knowledge base (username/password store), and the outcome of a successful authentication (which principals and credentials are added to the subject). The AuthenticationEnforcer creates a new LoginContext instance and calls its login method. This creates a new Subject representing the authentication requester and passes it to a LoginModule implementation. For this example, JAAS framework provides a WindowsNTLoginModule. It implements two methods—the login method decides on the authentication, and the commit method populates the subject with appropriate principals and credentials when the authentication is successful. User inputs are collected through the implementers of the CallbackHandler interface. The program transformation introduces a default strategy which will most likely be modified. Alice can apply an Add Strategy [116] refactoring and add a custom implementation later. Similarly, the authentication algorithm in a customized implementation of the LoginModule interface can be a Strategy [74] that developers can extend. The Add Authentication Enforcer transformation provides a default implementation only; Alice then customizes to fit the application specific requirements.

Towards an Add Authentication Enforcer Tool

jGuard [112] is a Java security framework based on JAAS. This framework is for resolving access control problems in web and standalone applications. jGuard creates authentication component based on user specification. However, specifying jGuard is complicated because it requires a lot of parameters. Reducing the amount of specification required makes it simpler to use, but then it will create a generalized authentication that will have to be customized further. Usability studies can determine the optimum amount of specification.

A tool for an Add Authentication Enforcer transformation composes classes of an existing framework. The default authentication component must be customized by developers to fit their application. The developers also have to figure out how to hook the authentication component with their application. Applying Single Access Point (section 6.2.1) and Policy Enforcement Point (section 6.2.2) transformations beforehand helps, but the typical scenario is less ideal and requires more manual involvement.
8.4 Limiting Authentication Attempts to Prevent Brute Force Attacks

8.4.1 Add Account Lockout Transformation

Attackers try to guess users’ passwords by attempting to log in repeatedly. You want to save your application from these brute force attacks.

Limit the number of login attempts. Lock an account after a pre-defined number of invalid login attempts, and release the account only after manual intervention.

Motivation

Every application with an authentication feature faces attack attempts to bypass authentication. The most common method is to try to guess the authentication token of a user (typically a password). The users choose weak passwords, which further complicates the scenario. Nearly 40% of user-chosen passwords are readily guessable by a brute force attack [143]. Attackers have password cracking programs readily available that launch brute force attacks using a dictionary; some sophisticated cracking programs even accept a user’s personal information [152].

Educating user’s to adopt a strong password defends against password guessing attempts, but a more pro-active approach is to put some limit in the authentication component of a system to thwart the brute force password guessing attempts.

 Preconditions

Any program that performs authentication should put a limit on the number of unsuccessful authentication attempts.

 Mechanism

A developer specifies the authentication component where the lockout policy is enforced.

The transformation inserts the check; it may be as simple as adding a for loop with conditional clause.
Example

Suppose, Alice has a Java program that connects to a hostname and a port via a socket. Alice has applied the Add Authentication Enforcer transformation (section 8.3.1) to create an authentication component composing classes of the JAAS framework. Figure 8.1 and the example section of the Add Authentication Enforcer transformation (section 8.3.1) describes the classes composed by the transformation.

The authentication enforcer component creates an instance of LoginContext class and invokes its login method to start the authentication process. Alice identifies the place in the program where the login method is invoked, and applies the Add Account Lockout transformation. The transformation introduces a for loop that keeps track of the number of login attempts and if clamps the number of login attempt to a specified number.

Towards an Add Account Lockout Tool

This program transformation is manual. The code that imposes the limit check should be introduced manually; after all, the limit only has to be introduced in a few places.

If a developer has applied an Add Authentication Enforcer (section 8.3.1) transformation to create a standard authentication component, then a tool can be introduces that introduces a code stub around the call of authentication attempt to limit it to some set limit. But finding where to insert the code is difficult if the authentication component is implemented arbitrarily.

In all cases, the program transformation requires few changes to the code. The harder part is to find where to introduce the code. Program analysis tools can be built to find the places that need to be changed.

8.5 Introducing Single Sign On Components

8.5.1 Credential Tokenizer Transformation

Software applications use various types of security tokens. You want to implement single sign on among these diverse systems so that they communicate and exchange tokens between themselves seamlessly.

Create a component that encapsulates different types of user credentials as a portable security token. The component provides a security API abstraction that creates and retrieves the user identity information from a given user credential.
Motivation

Security tokens can represent various types of user credentials, e.g. username/password, X.509v3 certificate, Kerberos ticket etc. Systems that want to communicate and use a single sign on service may be hindered by their use of different user credentials. To make federation possible in such a scenario, developers have to modify the security token handling functions, but there is no standard way to do so.

A simpler alternative is to create a separate credential processing component that encodes and decodes different credentials into security tokens.

 Preconditions

A program with the following characteristics benefits from a message Credential Tokenizer transformation

1. There are multiple programs communicating and they use different user credentials.

2. The program has gone through Single Access Point (section 6.2.1) and Policy Enforcement Point (section 6.2.2) transformations and it is possible to identify the input channels through which the security tokens come in.

3. A developer wants to create a separate component for processing security tokens instead of spreading that logic in the business logic or the authentication process.

Mechanism

A developer specifies the token types, the service, authentication scheme, and underlying protocol bindings.

The transformation creates a tokenizer that creates and manages the security token. It composes classes of a security framework to introduce a default component to encapsulate credentials as a standard token. A credential tokenizer provides an API abstraction for creating security tokens from user credentials to be presented to an authentication mechanism. It also provides API for retrieving security tokens issued by a security infrastructure provider.
Example

Suppose Alice has a Java application that uses username and password tokens. A Credential Tokenizer transformation creates a component for her that creates these tokens composing Java classes. Figure 8.2 shows the relevant classes that are generated. A CredentialTokenizer class uses an implementation of the TokenContext interface to create a UsernameToken.

```
interface TokenContext

- setType(tokenType: String): void
- createToken(principal: String, securityToken: Object): void
- getToken(): Object
- getPrincipal(): String
- getProtocolBinding(): String
```

```
interface SecurityToken

- #element: org.wkc.dom.Element
- getPrincipal(): String
```

```
interface UsernameToken

- getPassword(): String
- getToken(): String
```

![Diagram showing classes added by a Credential Tokenizer transformation]

Figure 8.2: Classes added by a Credential Tokenizer transformation

The CredentialTokenizer class delegates credential creation to a TokenContextImpl. It creates an instance of the username token and returns the token in XML.

Towards a Credential Tokenizer Tool

A Credential Tokenizer transformation is an Abstract Factory [74]. It generates a bunch of classes based on user specification. The description of the Credential Tokenizer pattern [194] includes more details of the component to be generated by the transformation tool.

The credential tokenizer thus implemented is a default tokenizer component. Developers can further extend it to fit their application needs.
8.5.2 Secure Session Object Transformation

In a distributed system, components need to have access to the security context of a client, otherwise clients have to authenticate for every request that they make. You want to save and route the context securely in order to enable single sign on.

*Encapsulate authentication and authorization credentials into a session object. Make sure the session object does not inadvertently expose data to non-privileged entities.*

Motivation

A distributed application needs to store, manage and distribute a security context associated with a client session. Typical web applications use cookies, and URL rewriting to store session information but there are security, privacy and performance implications. Using a Secure Session Object [194] for encapsulating authentication and authorization information allows distributed components to share security context in a secure manner. A developer needs to implement a standardized structure and interface to the security context.

Preconditions

A program with the following characteristics benefits from a Secure Session Object transformation

1. The application is in a multi-user distributed environment where a server application needs to keep track of client sessions.

2. A program uses cookies to store session state, but it raises security and privacy concerns.

Mechanism

A developer specifies what to put inside a session object, e.g. credentials, roles and privileges.

The transformation creates and manages a session object. The session object is shared between a server and a client. When a client authenticates, a server will create a secure session object that encapsulates user credentials and pass it to the client. The client will then pass the session object with each each request, and
the server will verify before processing a request. The program transformation will introduce mechanism to create and verify session objects.

Example

Suppose Alice wants to develop a client/server application where the session information is shared. The session information includes the credentials. She applies the Secure Session Object transformation to create a session object. The session object uses a Credential Tokenizer [194] (possibly created with a Credential Tokenizer transformation described in section 8.5.1) to encapsulate the credential. The transformation also introduces the mechanism to share the session object as a member of a more generic transfer object. The transfer object, created by the transformation, is a container for protecting session information. Two strategies for protecting content are to use a masked list or to use encryption. In each case, the developer has to specify additional details, e.g. encryption method etc.

Towards a Secure Session Object Tool

A Secure Session Object transformation composes a framework for creating and securely sharing session information. The steps of the transformation are complicated and mostly manual. Tools can be introduced to compose classes and create a default secure session object component. Even then, developers have to manually hook it to the application. Furthermore, developers have to extend it to fit their application needs.

Ideally, this transformation can be applied after a Credential Tokenizer transformation (section 8.5.1) has created a tokenizer object. The session object is shared as an instance of a Transfer Object [5]. Finally, the Obfuscated Transfer Object [194] pattern describes how the content of a transfer object can be protected.

8.5.3 Add Password Synchronizer Transformation

Different systems that are participating in a single sign on federation may have different user account management systems. You want to implement single sign on feature among these diverse systems.

Centralize user account management to a separate component that acts as a hub. Clients submit password service command to the hub which then issues service commands to all the application systems connected.
Motivation

Applications have diverse user account management mechanism. Administering the account management is very difficult if one has to consider all types of application systems. Same difficulty arises in managing user credentials (certificates, smart card tokens or even biometric samples) for authentication and authorization. Password has to be synchronized to be usable between systems. This becomes a more important issue in case of single sign on. A user wants to use passwords between multiple systems and wants to ensure that the systems do not have outdated credentials.

Preconditions

A program with the following characteristics benefits from an Add Password Synchronizer transformation

1. A group of applications each have their own different account management system.
2. A user wants to have single sign on feature.
3. It is possible to describe the interfaces of applications that share a password synchronizer component.

Mechanism

A developer identifies the systems that use a password synchronization service; he (or she) describes the account management interfaces of the identified services.

The transformations creates the password synchronizer service as a hub between all the components.

Example

Suppose Alice is a developer of two services and her company also uses two other services. She wants to implement a password synchronization service as a hub between the four services. When she applies the Add Password Synchronizer transformation, a new service is created that provides a façade for password management: clients use this synchronization service instead of directly accessing the four services. Alice has to provide a mapping that describes the interfaces of account management components of the four original services. Figure 8.3 shows the synchronizer component.
The password synchronizer service uses a ledger component to keep track of the service requests. Once the request is completed, the ledger marks the service as successful. This becomes useful when the target service is unavailable, and a request is retried; the ledger then keeps track of the pending requests.

Towards an Add Password Synchronizer Tool

The implementation section of the Password Synchronizer pattern [194] describes the password synchronizer component in detail. Because this transformation is very complex, it is probably impossible to automate the entire transformation. But parts of it can be automated, e.g., creating a password synchronizer component as a Mediator [74] of that knows the target service interfaces, creating a component that handles a mapping table, creating a ledger component etc.

8.5.4 Single Sign On Delegate Transformation

You want to implement a loosely coupled single sign on mechanism that interacts between diverse entities.

Encapsulate access to identity management and SSO functionalities. Create a separate delegator component for single sign on.
Motivation

Single sign on is implemented between services with heterogeneous security environments. Services for identity management may be implemented by multiple vendors and each require different service invocation methods and programming models. Designing a system where clients directly access remote identity management and single sign on interfaces is a bad idea, because it introduces tight coupling and exposes internal components to clients.

A system should encapsulate internal details of identity management and single sign on. If a separate delegator component is used, then this decoupling also makes changing identity management or single sign on security services easy.

Preconditions

A program with the following characteristics benefits from an SSO delegate transformation

1. A group of applications with heterogeneous security environments need to have single sign on.
2. The applications follow standard frameworks for implementing the identity management services.

Mechanism

A developer describes the services that share single sign on mechanism.

The transformation automatically creates an SSO delegator with default services. An SSO Delegator resides in the middle tier between the clients and the identity management service components. It delegates the service request to remote service components. Decouple the physical security service interfaces and hide the details of service invocation, retrieval of security configuration or credential token processing from the client. The SSO Delegator in turn prepares for SSO, configures the security session, looks up physical security service interfaces, invokes appropriate service and performs global logout in the end. Such loosely coupled application architecture minimizes the change impact to the client.

Example

An SSO Delegator is a hub that intercepts all requests from clients and delegates it to target services. Suppose Alice wants to apply an Single Sign On Delegate transformation on a group of Java applications. The
transformation composes Java classes to introduce the delegator component. Figure 8.4 shows the important classes of the delegator component.

![Diagram](image)

Figure 8.4: Important classes of the delegator component

The SSOContext encapsulates service configuration and protocol bindings to invoke remote service providers. The SSOServiceProvider is a public interface for creating, closing, or reconnecting to a remote service. Finally, the SSODElegatorFactory defines the public interfaces for creating and closing a secure connection. It uses a security token provided by the client.

When a client requests to a SSODElegator with a security token, SSODElegator first authorizes the client. Then it retrieves service configuration, and retrieves protocol bindings from an SSOContext object. SSODElegator then creates a service connection and invokes the remote service provider through the SSOServiceProvider interface.

**Towards a Single Sign On Delegate Tool**

The implementation section of the Single Sign On Delegate pattern [194] describes the delegate component in detail. Because this transformation is very complex, it is probably impossible to automate the entire transformation. But parts of it can be automated. For example, the task of looking up a service location and invoke it can be automated following the Service Locator pattern [5]. Also, the user credentials handled by the SSO delegator can be managed by a credential tokenizer component, created by applying a Credential Tokenizer transformation (section 8.5.1). In general, the parts that are automated are primarily the parts that handle delegation; they essentially replace one component with another.

141
8.6 Organizing the Transformations

Most program transformations described in this chapter add large authentication components and delegate the task. In some cases, the components span multiple system boundaries.

The program transformations are organized using two criteria: their impact on codebase and the type of transformation. Table 8.1 describes how they are organized.

<table>
<thead>
<tr>
<th>Impact on Codebase</th>
<th>Type of Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small, frequent change</td>
<td>Add Authentication Enforcer</td>
</tr>
<tr>
<td>Large change</td>
<td>Add Account Lockout</td>
</tr>
<tr>
<td>Change beyond boundaries</td>
<td>Add Session Object</td>
</tr>
<tr>
<td></td>
<td>Add Credential Tokenizer</td>
</tr>
<tr>
<td></td>
<td>Add Single Sign On Delegate</td>
</tr>
<tr>
<td></td>
<td>Add Password Synchronizer</td>
</tr>
</tbody>
</table>

Table 8.1: Security-oriented Program Transformations for Broken Authentication

8.6.1 Impact on Codebase

Add Authentication Enforcer and Credential Tokenizer transformations make large change in the code, while Add Account Lockout transformation only makes a few changes in a small portion of the code. The remaining three program transformations changes multiple applications. Single sign on involves many entities. As such, a program transformation that adds single sign on components makes change in more than one system.

8.6.2 Type of Transformation

Most program transformations described in this chapter add new components. Only Add Add Account Lockout transformation imposes a limit on the number of authentication attempts.

Add component. Authentication components are added at the entry point of a system. A program transformation adds a component and delegates a task to it. For example, an authentication component based on JAAS framework creates a login module that takes the login decision. A developer has to specify how the credentials, e.g. username and password, are passed to the login module. A login module decides on the
authentication, and initializes a subject. It populates a subject with principals and credentials that is used by authorization components at different points in the system.

Large program transformations require large user specifications. For an Add Authentication Enforcer transformation that creates a username and password authentication component, a developer has to specify where the inputs are coming from, where and how a system stores username password pairs, how the authentication decision is made, and what principals and credentials to add given an authentication attempt is successful. It is difficult for a user to provide all these specifications for an automated tool; at the same time parsing a complex specification will make a tool complex. A simpler alternative for a tool will be to create a default authentication component and let the developers customize it for their specific needs. This can be done by creating hook methods, or by adding a Strategy [74]. Frameworks such as JGuard compose authentication components based on user-provided specification.

Limit. Even solid authentication mechanisms can be undermined by flawed credential management functions, such as password change. An Add Account Lockout transformation adds a limit on the number of login attempts. Unless the authentication component is created following a standard framework (JAAS or .NET), it is difficult to add this check automatically. However, since it is only done at a few places of a system, it can be done manually.

### 8.7 Composing the Transformations

An Add Authentication Enforcer transformation usually follows a Single Access Point (section 6.2.1) and Policy Enforcement Point transformation (section 6.2.2). A Policy Enforcement Point transformation creates a policy enforcement component and a placeholder authentication component. An Add Authentication Enforcer transformation replaces the placeholder component.

An Add Account Lockout transformation is usually applied after an Add Authentication Enforcer transformation to impose the limit in the appropriate place of the standard authentication component.

Among the 4 transformations for enabling single sign on, a Credential Tokenizer transformation is usually applied first to create a tokenizer component. This component is used by the Secure Session Object and Single Sign On Delegate transformations. All four transformations are very large and hard to automate, but parts of them can be automated. They all require that the identity management features are implemented in
a systematic manner.

8.8 Summary

Many security problems that are under the category of unvalidated input or broken access control are in fact problems with the authentication mechanism of a system. Bad authentication allows an intruder in, who can then take advantage of other vulnerabilities in the system.

This chapter describes 6 security-oriented program transformations. These program transformations add authentication components composing classes from authentication frameworks.
Chapter 9

Security-oriented Program Transformations for Access Control

Authorization or access control regulates how an application grants access to resources to users with sufficient privilege. A lot of access control errors originate from authentication, as attackers manage to trick authentication step and gain improper privilege level. But even if the authentication step works correctly, attackers can gain improper privilege if the access control mechanism is not implemented correctly.

After a successful authentication step, a subject is created that represents a user. The resources to which access may need to be controlled are called objects. Access control policies describe which subjects are allowed to access which objects. The task of an authorization or access control component is to enforce access control policies during resource access.

This chapter considers access control from a broader perspective and describes 7 security-oriented program transformations that facilitate various aspects of access control. They cover classic authorization problems and how authorization components can be introduced systematically. Additionally, they cover how access rights are created during a process or an object creation, and how can they be regulated. They also cover how resources can be protected by introducing cryptographic methods such as encryption, signature etc.

9.1 Broken Access Control Vulnerability

OWASP list of top ten vulnerabilities [151] catalogs authorization problems under the ‘broken access control vulnerability’ category. Authorization checks are performed after authentication, and govern what ‘authorized users are allowed to do. We extend this definition of access control and include issues regarding privilege handling, protection of resources etc. The focus of this chapter is on 3 access control issues faced by developers.

First, developers frequently underestimate the difficulty of implementing a reliable authorization mech-
anism. They add access control policies in an ad hoc manner as a system evolves, instead of designing an authorization mechanism. A symptom of this lack of conceptual integrity is the presence of authorization mechanism in various locations all over the code. Authorization policies may change as a system evolves, but the mechanism of handling authorization should be present in a systematic manner. Frequently, these issues are intertwined with the authentication issues. We have surveyed the Bugtraq vulnerability list [34] on several random weeks and found that on average 5% of the attacks are related to broken authentication and broken access control vulnerability.

Second, programming platforms make it hard to for developers to control privilege. When a Unix child process is forked, it inherits all the privileges from parents, unless a developer explicitly intervenes. Again, this is a manual, ad hoc process that often leads to many mistakes.

Third, applications need mechanism to protect stored and transmitted data. Protection not only involves concealing data from unauthorized access, but also ensuring the integrity of data. Typically developers use cryptographic APIs and implement these features manually, but their implementation often becomes a weak cryptographic solution because of improper use of the API. Another type of protection is protecting data in transit, to retain the privacy of transmitted data. An attacker can observe a data in transit and identify the source and destination of an exchange. This can compromise the privacy of the sender and the receiver.

9.2 Solution Strategies

We describe 7 security-oriented program transformations to help developers mitigate with the issues mentioned in section 9.1. These program transformations help developers in three contexts: 1) when a developer wants to introduce a new authorization component, 2) when a developer wants to regulate how privilege is passed during process or object creation, and 3) when a developer wants to protect stored and transmitted data.

An Add Authorization Enforcer transformation introduces an authorization component. The authorization component intervenes when there is an attempt to access a resource, and only allows access to a privileged entity.

We describe 3 program transformations that regulate how privilege is calculated during a process (or an object) creation time. Controlled Process Creation transformation regulates how a parent process passes privilege to child processes; Guarded Object transformation encapsulates a newly created object with its
privileges so that authorization can be done in a different context. Secure Resource Pooling transformation creates a pool of pre-forked child processes and maintains the resource pool.

The remaining 3 program transformations are related to protecting stored and transmitted data. Encryption/Decryption, Message Digest Creation and Signature Generation transformations use cryptographic APIs to protect confidentiality and integrity of data.

9.3 Adding Authorization Components

9.3.1 Add Authorization Enforcer Transformation

How can you use a subject’s credentials to determine whether a user is allowed to access a resource? Add an authorization enforcer component that uses the permissions of principals to determine whether a subject is allowed to act upon a resource.

Motivation

The purpose of authenticating a user is to establish its identity so that its actions can be controlled. The subject created in the end of authentication represents a user. Each subject is associated with an access control context, i.e. every action happens as if a subject is running it. Each subject has many principals, each of which has permissions associated. When a subject attempts to access a resource, an authorization component has to check whether the subject has permissions to access the resource. The authorization component matches the permission requested with the permissions retrieved from a subject’s principals.

There are two aspects of authorization: the definition of policies and policy enforcement. Developers define the security policies that describe which accesses are allowed. The task of an authorization component is to enforce the policies defined by application developers. Concentrating authorization logic in one component makes it easy to maintain access control. The authorization component is an Interceptor [183] that intervenes all access requests.

Typically, developers create an authorization component by using an authorization framework suitable for their platform and composing classes of the framework. Much of this task can be automated.
Preconditions

A program with the following characteristics benefits from an *Add Authorization Enforcer* transformation.

1. The program has an authentication component that has been implemented using an authentication framework. An *Add Authentication Enforcer* (section 8.3.1) transformation can be used to create an authentication component.

2. A set of access control policies have been defined.

Mechanism

A developer specifies how to get an authenticated subject, the set of policies of a system, the action that needs to be authorized, and the authorization strategy.

The transformation composes classes of a security framework (JAAS, Java 2 Security API, .NET etc) to create a component with the specified authorization strategy, and wraps the action so that an access control decision is made before the action is performed.

Example

Suppose, Alice wants to add an authorization enforcer component to a Java program. Before applying the *Add Authorization Enforcer* transformation, Alice has applied the *Single Access Point* (section 6.2.1), *Policy Enforcement Point* (section 6.2.2), and *Add Authentication Enforcer* (section 8.3.1) transformations.

The authentication component verifies user identity and then creates a subject to represent a user.

Alice has two options for creating an authorization component in a Java based system: using JAAS principal based policy files, or using the permission hierarchy of Java 2 security API. Let us suppose that Alice wants to use JAAS authorization strategy.

Alice secures an action (e.g. a method call or a resource access) by applying the *Add Authorization Enforcer* transformation. Alice specifies the login context (*LoginContext* class created by an authentication component; an example is an authentication component created by the *Add Authentication Enforcer* transformation described in section 8.3.1), authorization strategy (JAAS principal based authorization), the set of policies (location of policy file to be used by the JAAS framework), and the action that needs to be
secured (a portion of the code). The transformation wraps the action as a privileged action and only allows it if the subject has sufficient permission.

Figure 9.1 shows the steps of authorization and the classes involved. The AuthorizationEnforcer class retrieves the Subject from the LoginContext, wraps the portion of code to protect with a new instance of PrivilegedAction, and invokes the doAsPrivileged method of the Subject to perform the PrivilegedAction. The doAsPrivileged method uses JAAS framework to make the authorization decision and runs the PrivilegedAction if authorization is successful. The JAAS framework classes use the policy specified by Alice to make the decision. The transformation creates a default implementation based on user specification; a developer may have to customize further to meet application requirements.

**Towards an Authorization Enforcer Tool**

The Add Authorization Enforcer transformation tool creates a default authorization component. The component is generated by using classes from a framework. Developers can further extend it to fit their application needs. The description of the Authorization Enforcer pattern [194] includes more details of the component to be generated by the transformation tool.
9.4 Regulating How Privilege is Passed during Process or Object Creation

9.4.1 Controlled Process Creation Transformation

When a parent process spawns a child process, it shares data and system objects with the newly spawned process. This creates a risk that sensitive data or system objects may be leaked between a trusted parent and an untrusted child process. You want to prevent this type of privilege escalation during process creation.

Explicitly control data and shared objects that are passed between a parent process and a newly spawned process.

Motivation

A parent and a child process share various information: memory, file system information, open file descriptors, environment variables, memory mappings, signals, semaphores etc [149]. If parent and child processes run with same privilege, these shared channels do not create a serious security problem. However, often parent and child process hierarchy is used to separate trusted and untrusted processes; untrusted processes are typically spawned from trusted ones. Unless the process creation channel is explicitly controlled, there is a potential privilege escalation problem.

Preconditions

A program with the following characteristics benefits from a Controlled Process Creation transformation

1. The program uses system calls, such as fork, to create child processes.
2. The parent process and the child process have different access rights.

Mechanism

A developer specifies the parent and child processes by identifying the process creation system calls.

The transformation analyzes program for sensitive data and file system descriptor leaks, and allows a developer to select which data and file systems can be shared. Upon user specification, the program transformation adds system calls to initialize child process environment, ensures that sensitive file descriptors
are closed, and moves code so that sensitive data is not shared. Thus, there is no automatic inheritance of rights by child processes.

Example

Consider a \texttt{fork} call in Linux. The parent process that calls \texttt{fork} shares various data and system object. Some of these shares cannot be moderated, e.g. code memory, file system information, process memory mapping etc. Code memory can usually be derived from source code of the child process, whereas file system information and process memory mapping information are available via the \texttt{/proc} filesystem.

On the other hand, data memory, open file system descriptors, shared memory, environment variables etc. are shared unless explicitly controlled. A \textit{Controlled Process Creation} transformation applied to \texttt{fork} makes various changes.

Adding \texttt{exec} call. Since \texttt{exec} call and its variants clear data (and stack) memory and shared memory of child processes, the program transformation adds this call during initializing a child process. An \texttt{execve} call, a variant of \texttt{exec}, also initializes environment variables. In order to facilitate the \texttt{exec} call, some code movement may be necessary.

An \texttt{exec} call replaces the current process image with a new process image. The new image is constructed from a regular, executable file called the new process image file. But in the original parent process, the function run by a child process may be in the same file. In order to introduce the \texttt{exec} call, the code run by the child process has to be moved to a new file. Consider the following example \texttt{fork} taken from Orion Lawlor’s short Unix examples repository located in \url{http://lawlor.cs.uaf.edu/~olawlor/ref/examples/unix/index.html}.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h> /* for pid_t */
#include <sys/wait.h> /* for wait */

void * doWork(const char * where){
    int i;
    int b=1,a=1;
    printf("Entered work function (%s)\n",where);
    fflush(stdout);
    for (i=0;i<100000000;i++)
        b=a+b;
    printf("Leaving work function (%s)\n",where);
    return (void *)b;
}
```

151
int main()
{
  /* Spawn a new process to run alongside us. */
  pid_t pid=fork();
  if (pid==0) { /* child process */
    doWork("child");
    exit(0);
  }
  else { /* pid!=0; parent process */
    doWork("parent");
    waitpid(pid,0,0); /* wait for child to exit */
  }
  return 0;
}

The child process executes the doWork function in line 23. In order to run the child process with an exec call, a Controlled Process Creation transformation moves child process code from the source file to a new file. The following listing is of the new source file for the child process. A new main function is created that calls the doWork function.

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h> /* for pid_t */
#include <sys/wait.h> /* for wait */

void * doWork(const char * where){
  int i;
  int b=1,a=1;
  printf("Entered work function (%s)\n",where);
  fflush(stdout);
  for (i=0;i<100000000;i++)
    b=a+b;
  printf("Leaving work function (%s)\n",where);
  return (void *)b;
}

int main(int argc, char * argv[]){
  doWork(argv[1]);
  exit(0);
}

The parent process is modified in line 23. Instead of directly calling the doWork function, the parent process marshals data in an argument list and calls exec. The following shows the modified portion only.

... if (pid==0) { /* child process */
  static char * argv[]={"forkchild","child",NULL};
  execv("forkchild",argv);
  exit(127); /* only if execv fails */
} ...

152
**Closing open file descriptors.** Open file descriptors are shared by a `fork` call, even though `exec` is called. In order to ensure that open file descriptors are not passed, sensitive file descriptors should be associated with `FD_CLOEXEC` flag. This will force an `exec` call to close the open file descriptor.

A *Controlled Process Creation* transformation finds the open file descriptors, typically by searching the `/proc` file hierarchy. A developer, upon viewing this list, may determine some of the files as sensitive. The program transformation then associates `FD_CLOEXEC` flag with those file descriptors.

```c
... 
fcntl(fd, F_SETFD, FD_CLOEXEC);
...
```

**Towards a Controlled Process Creation Tool**

A *Controlled Process Creation* transformation moves code blocks and adds system calls to control information flow across a process creation call. Creating a tool that analyzes control flow of a program and performs these tasks automatically is not difficult.

Additionally, a tool may allow a developer to identify sensitive data that are leaked from a parent process to a child. This combines control-flow and data-flow analysis. A developer selects the data variables that he or she considers sensitive, and a tool analyzes a program to determine the leakage. Shankar et. al. [188] describe this program analysis algorithm in detail.

**9.4.2 Guarded Object Transformation**

You want to design a system that can take access control decisions locally.

*Encapsulate resources with access control context so that access control decisions can be taken locally.*

**Motivation**

When a system makes access control decisions, it relies on the context information. However, the information may not be present if the access control decision is to be made in a separate context, e.g. the supplier and the consumer of a resource are in different threads. If the context becomes a part of the protected resource, then taking access control decisions becomes easier.
Preconditions

A program with the following characteristics benefits from a Guarded Object transformation

1. The program can use a security platform that provides an infrastructure to create a Reference Monitor [184].

2. The program has an authentication infrastructure that creates a subject upon successful authentication. Each subject has a list of permissions. An Add Authentication Enforcer (section 8.3.1) transformation can be used to create an authentication component.

Mechanism

A developer specifies a resource to be protected and the permissions associated with the resource.

The transformation creates a guarded object that encapsulates a resource and the permissions associated with the resource. It also changes all the access instances of the unprotected resource with a call to the guarded object that returns the resource. The guarded object decides whether the caller has sufficient permission to access the resource. The decision is based on the policy information.

Example

Suppose Alice is a Java developer. She wants to associate a ‘read’ permission to a FileInputStream object and create a guarded object. The guarded object protects access to the FileInputStream object and only allows access to subjects with read permission. Figure 9.2 is class diagram of the relevant classes generated by a Guarded Object transformation; it shows how a GuardObject can be introduced to wrap an UnprotectedResource.

FileInputStream will be encapsulated into a GuardedObject, and the actual object is returned every time it is accessed. The Java security architecture ensures that the permission is checked.

Suppose Alice has a Guarded Object transformation tool available as an Eclipse plugin. She selects the definition of FileInputStream object in her Eclipse IDE and applies the transformation.

```java
FileInputStream fs = new FileInputStream (/home/munawar/abc.txt);
...
fs.read (...);
```
Figure 9.2: A guard object encapsulating an unprotected resource as created by an *Guarded Object* transformation

The transformation tool asks for permissions to be associated with the file. Alice selects `FilePermission` and it has the read property. The following changes are made to the code.

```java
FileInputStream fs = new FileInputStream (/home/munawar/abc.txt);
FilePermission perm = new FilePermission (/home/munawar/abc.txt, read);
GuardedObject g = new GuardedObject (fs, perm);
...
FileInputStream fis = (FileInputStream) g.getObject();
fis.read (...);
...
```

A developer can add custom permissions by extending the `Permission` class and implementing the `implies` and `getActions` methods.

**Towards a Guarded Object Tool**

A *Guarded Object* transformation composes classes of a security framework and creates a *Reference Monitor* [184]. Its implementation is similar to a *Decorated Filter* transformation (section 7.4.1), which decorates an input variable with input validation filters.

A *Guarded Object* tool encapsulates an object with associated permissions. A tool uses the permissions described by the security framework. However, it is possible to extend the list of permissions with new custom permissions suitable for a particular domain.
9.4.3 Secure Resource Pooling Transformation

The consequences of security compromise are worse for daemon processes compared to processes with limited lifetime. An attacker can exploit the long lifetime to compromise other processes. You want to prevent attackers from exploiting the long lifetime of daemons.

*Limit the lifetime of daemon processes and fork them again after a configurable, short lifetime. Create a monitor that monitors process pool and replenishes based on user configuration.*

Motivation

Network servers handling concurrent requests use resource pooling to improve performance. A simple example is a master server process acting as a gatekeeper to the system. It waits for a connection request and when a request arrives, it forks itself to create a child process. Since process forking is expensive, the master server pre-forks a pool of child processes ready to handle incoming clients.

The pre-forked processes have a long lifetime. The consequences of a security compromise with these processes can be severe, because the processes give the attacker a long time to exploit other processes.

The solution is to limit the lifetime of daemon processes [84]. A monitor process checks the process pool and kills child processes. Which process is to be killed depends on user specified parameters. The identified process is killed and a new process is forked in the process pool in its place.

Apache uses this approach. Apache has various configuration parameters that are used during server startup and process management. Apache master server uses the startup parameters to spawn off a number of child processes in the resource pool. The master server maintains memory resident data structures for process accounting. Based on these values, it replaces child processes after their limited lifetime.

Preconditions

A program with the following characteristics benefits from a Secure Resource Pooling transformation:

1. The program is a network server that handles concurrent requests. It uses resource pooling to improve performance.

2. The processes in the resource pool are daemon processes.
Mechanism

A developer specifies the parameters that determine the lifetime of worker processes in a resource pool. The transformation creates a monitor process for the resource pool that kills processes in the pool and replenishes it with new processes.

Example

Suppose, Alice has a network server program that forks child processes on demand. Alice applies the Secure Resource Pooling transformation to create a resource pool.

She specifies various parameters to specify the resource pool. Some of these parameters determine how the resource pool is initialized, e.g. how many child processes to fork on startup. Other parameters define how the monitor process keeps track of child processes. Examples of such parameters include the maximum time that a process remains idle, the maximum number of requests that a process serves, the maximum life limit of a process since its creation, etc. The program transformation creates a monitor component that enforces the policies.

Towards a Secure Resource Pooling Tool

The hard part of a Secure Resource Pooling transformation is to find which places in the program need to be changed. Once that is done, the change is easy.

Secure Resource Pooling transformation composes a default component that includes a monitor process and a resource pool, and then customizes the component based on the parameter specified by Alice. The task can be automated since it involves customizing a default component with parameters.

The program transformation also delegates all the child process creation calls (e.g. fork) to the monitor process. The monitor process, instead of creating a new process, takes a process from the resource pool, redirects the I/O of it, and attaches it to the parent process. Also the exit calls of child processes are delegated; in this case, the monitor process intercepts these calls, reclaims the child process, updates information about the child, and notifies the parent.

Identifying the right parameters is a manual process, but there should be some guidelines. The most important issue is to determine the criteria for process killing. An obvious parameter is the number of requests served by a process or the duration of service. However, this parameter alone is insufficient as
attackers are tempted to launch their attacks during low-traffic or off-peak hours. Hence the idle time of processes should also be considered. Another related parameter is the overall lifetime of processes. However, this parameter cannot be used alone. Since processes are pre-forked as a group, their overall lifetime will reach a threshold together; but killing them all has performance penalties. The key is to consider multiple parameters.

Also important are the initialization parameters. Consider the number of initial pre-forked child processes in the resource pool. If the count is too high, the server startup time is increased. On the other hand, a low count means that the server may have to spend time in spawning new processes if the initial load is high. The minimum and maximum value of parameters are set according to some empirical values seen from trial runs of the software.

The pre-forking parameters can also be dynamically determined. A smart pre-forking approach can initialize processes based on load. The system can use an algorithm that incrementally increases the number of processes spawned dynamically in response to increasing load. An example of this is in Apache implementation [82]. Apache has a responsive algorithm for handling incoming requests: initially, it does not pre-fork the total number of required processes. Instead, it continues to spawn between 1 to 32 new processes per second, until all clients are satisfied. The objective of this is to prevent Apache from starting up excessive numbers of processes all at once unless it is actually necessary. Only if Apache is genuinely experiencing a sharp rise in demand will it spawn new processes after initialization. Apache’s smart and dynamic handling of the server pool makes it capable of handling large swings in demand.

Several patterns describe the implementation mechanism of a Secure Resource Pooling tool. The Pooling [120] pattern focuses on the pre-forking and resource pool creation issues. The Resource Lifecycle Manager [120] pattern describes how the monitor process is created and how the resource management task can be decoupled. Another important aspect of the secure pre-forking pattern is the release of resources. The Evictor [120] pattern describes how and when to release resources to optimize resource management.
9.5 Protecting Resources

9.5.1 Encryption/Decryption Transformation

You want to protect stored data and message in transit so that unauthorized users cannot view the content.

*Encrypt data when it is stored or in transit. Decrypt data before it is used.*

Motivation

A secure application should protect its sensitive data at all times, typically by using encryption mechanism. If stored data is not encrypted, an attacker can have access to stored contents after compromising a system. Similarly, data in transit must be encrypted to protect from eavesdroppers.

Encryption is a dual mechanism. Encrypted data should be decrypted before use.

Preconditions

A program with the following characteristics benefits from an encryption/decryption transformation

1. Data is encrypted using standard encryption algorithms.
2. The program uses a cryptographic framework to encrypt/decrypt data.

Mechanism

A developer specifies the input data buffer, the parameters of the encryption algorithm, the key used to encrypt/decrypt a message, and the key encoding algorithm.

The transformation composes classes from a security framework (JCA, JCE, .NET etc) to encrypt/decrypt the message.

Example

Suppose Alice has a network based Java application. She wants to add encryption mechanism to secure all messages in transit. She selects a `String` variable that represents a message and applies an Encryption/Decryption transformation.
Alice has to specify a number of parameters. Alice specifies the encryption algorithm, e.g. DES [6] in electronic code book (ECB) mode [176]. Since DES is a block cipher, Alice has to choose a padding scheme; suppose she chooses PKCS#5 padding scheme [177]. She also specifies the source of the cryptographic key that is to be used. The following lists the code generated by the program transformation. In this example, the encryption code is composed from JCE framework.

```java
Cipher myCipher = Cipher.getInstance ("DES/ECB/PKCS5Padding");
myCipher.init(Cipher.ENCRYPT_MODE, key);
String message = ... ;
byte[] myciphertext = myCipher.doFinal (message.getBytes());
String encryptedMessage = getString (myciphertext);
```

The program transformation also replaces the instances of `message` with `encryptedMessage`. In the example, Alice assumes that the key is already available. If a key needs to be created, the program transformation can also generate code for key initialization.

### Towards an Encryption/Decryption Tool

An *Encryption/Decryption* transformation tool composes classes from a security framework (JCA, JCE, .NET etc) based on user specification.

It can replace the variable, that is to be encrypted or decrypted, with another variable instance. Or it can use the *Decorated Filter* transformation (see section 7.4.1), and apply encryption/decryption as a filter.

#### 9.5.2 Message Digest Creation Transformation

You want to ensure that an active attacker cannot corrupt stored or transmitted data.

*Calculate digest and add it to data to prove integrity.*
**Motivation**

A message digest is a one-way hash function that verifies the integrity of data. Data producer calculates the digest, signs the digest, and adds it with data. Even if an attacker corrupts data, he or she cannot modify the signed digest.

Additionally, message digests can be used for secret sharing. For example, instead of sending a password to an authentication engine, a user may pad an agreed upon nonce with password, and send the digest of the resultant data. A passive observer cannot find out what the password is. Moreover, since a digests is a one way hash, a brute force attacker faces the same cryptographic difficulty of guessing the password.

** Preconditions**

A program with the following characteristics benefits from a message digest transformation

1. Message digest is calculated using standard encryption algorithms.
2. The program uses a cryptographic framework to create message digest.

**Mechanism**

A developer specifies the input data buffer, and the message digest algorithm to be used.

The transformation composes classes from a security framework (JCA, JCE, .NET etc) to calculate message digest.

**Example**

Suppose Alice has a network based Java application. She wants to add message digest to all outgoing messages. She selects a `String` variable that represents a message and applies a `Message Digest Creation` transformation.

```java
...
String message = ...;
...
```

Alice specifies the algorithm used for creating message digest. For this example, Alice chooses SHA-1 algorithm [58].
A Message Digest Creation transformation tool generates the message digest. In this example, the message digest code is composed from JCA framework.

```java
MessageDigest sha = MessageDigest.getInstance (SHA-1);
String message = ... ;
sha.update (message.getBytes());
byte[] hashValue = sha.digest();
... 
```

Alice may need to create and initialize the key that will be used for message digest creation. In that case, she will have to specify a secure random number and initialize it with a seed value.

Towards a Message Digest Creation Tool

A Message Digest Creation transformation tool composes classes from a security framework (JCA, JCE, .NET etc) based on user specification. The message digest creation mechanism can be added to data as a Decorator [74] (see Decorated Filter transformation in section 7.4.1).

The number of algorithms for message digest creation is nominal compared to the number of algorithms for encryption/decryption or signature generation. For example, the J2SE JCA provider only supports two message digest algorithms: MD5 and SHA-1. Consequently, this tool is simpler to build compared to an Encryption/Decryption tool (see section 9.5.1) or a Signature Generation tool (see section 9.5.3).

9.5.3 Signature Generation Transformation

An entity responsible for data may later deny any responsibility of the content. You want to add non-repudiation mechanism to the data used in an application.

Sign data with a private key.

Motivation

A signature provides non-repudiation. The entity that generates a signature signs a document with its private key and sends it to the data consumer entity along with a certificate bearing its public key. The data consumer extracts the key from the certificate and uses it to verify the signature, thus getting assured that the data is...
coming from the signing party. A signature is the basis of trust: a data consumer entity trusts a data provider implies that it trusts a message generated and signed by a provider entity.

Signature generation is a dual mechanism. At the data provider end, signature is generated and added with data. At the consumer end, signature is verified before data is used.

Preconditions

A program with the following characteristics benefits from a signature generation transformation

1. Signature is generated using standard encryption algorithms.

2. The program uses a cryptographic framework to sign data.

Mechanism

A developer specifies the input data buffer, the parameters of the signature algorithm, the key used to sign a message, and the key encoding algorithm.

The transformation composes classes from a security framework (JCA, JCE, .NET etc) to generate message signature.

Example

Suppose Alice has a Java application. She wants to add signature mechanism to that the application signs all outgoing messages. She selects a String variable that represents a message and applies a Signature Generation transformation.

```java
String message = ...;
...
```

Alice specifies the location of the private key. If a private/public key pair is not present, Alice can also specify additional parameters to generate and initialize a key pair. This example assumes that a key pair is available. A Signature Generation transformation tool generates code that uses privateKey of the key pair to create a digital signature. In this example, the signature code is composed from JCA framework.
try {
    Signature dsig = Signature.getInstance (privateKey.getAlgorithm());
    String message = ... ;
    dsig.initSign (privateKey);
    dsig.update (message);
    byte[] signedData = dsig.sign();
} catch (SignatureException e) {
} catch (InvalidKeyException e) {
} catch (NoSuchAlgorithmException e) {
}

... 

Alice also specifies the public key certificate; the certificates is send along with the message.

At the receiver end, Alice applies the Signature Generation transformation. She specifies the data to be verified (signedData) and certificate from which the public key (publicKey) is retrieved. The following shows the code the the transformation introduces to verify the signature.

try {
    Signature dsig = Signature.getInstance (publicKey.getAlgorithm());
    dsig.initVerify (publicKey);
    dsig.update (signedData);
    boolean result = dsig.verify (signatureToVerify);
} catch (SignatureException e) {
} catch (InvalidKeyException e) {
} catch (NoSuchAlgorithmException e) {
}

... 

Towards a Signature Generation Tool

A Signature Generation transformation tool composes classes from a security framework (JCA, JCE, .NET etc) based on user specification. The message digest creation mechanism can be added to data as a Decorator [74] (see Decorated Filter transformation in section 7.4.1).

9.6 Organizing the Transformations

Program transformations that prevent access problems follow various strategies, e.g. introduce an authorization component, manage privilege level during process creation, or employ cryptographic techniques to keep data confidential.

The program transformations are organized using two criteria: their impact on codebase and the type of transformation. Table 9.1 describes how they are organized.
9.6.1 Impact on Codebase

Many members of this category are large transformations. Large transformations are harder to automate and they require more detailed specification from the developers. For an Add Authorization Enforcer transformation, a developer has to specify principals, requesting context, permissions, etc.

9.6.2 Type of Transformation

Add component. Adding an authorization component at resource access points involves composing a lot of classes. For example, an authorization component based on Java Authentication and Authorization Service (JAAS) framework creates a collection of permissions and stores it in the credential set of a subject. Composing an authorization component involves composing many classes, e.g. classes describing a subject, its principals, their permissions, permission collection, credentials, request context.

<table>
<thead>
<tr>
<th></th>
<th>Small, infrequent change</th>
<th>Small, frequent change</th>
<th>Large change</th>
<th>Change beyond boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Component</td>
<td>Guarded Object</td>
<td></td>
<td>Add Authorization Enforcer</td>
<td></td>
</tr>
<tr>
<td>Limit</td>
<td>Controlled Process Creation</td>
<td></td>
<td>Secure Resource Pooling</td>
<td></td>
</tr>
<tr>
<td>Mutate data</td>
<td>Message Digest Creation</td>
<td>Signature Generation</td>
<td>Encryption/ Decryption</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.1: Security-oriented Program Transformations for Broken Access Control

Mutate data. Another type of transformation that indirectly helps access control is to apply cryptographic operations to program data. For example, a developer may specify the algorithm used for calculating a message digest of a data value, and a Message Digest Creation transformation automatically introduces calls to a security API to calculate the digest.

Limit. Sometimes enforcing stricter constraints result in better access control. A Secure Resource Pooling transformation modifies the manager process of a resource pool to limit the lifetime of worker processes in the pool. A developer has to specify a policy that determines the lifetime of worker processes, e.g. number
of times a process can serve, number of times a process remains idle, total amount of time a process is in the pool etc. A tool can automatically create a manager process and add the policy.

9.7 Composing the Transformations

Add Authorization Enforcer and Guarded Object transformations usually follow an Add Authentication Enforcer transformation (see section 8.3.1). Add Authentication Enforcer transformation authenticates a user and creates a subject to represent the user. The subject has permissions; these permissions are used by the authorization component.

Encryption/Decryption, Message Digest Creation and Signature Generation transformations mutate data. They can be implemented as a Decorated Filter transformation (see section 7.4.1): each cryptographic operation is applied as a filter. However, the order in which the filters are applied is important with cryptographic operations. An input that has gone through encryption followed by the calculation of a message digest is to follow a reverse order when it is decrypted.

9.8 Summary

Broken access control vulnerability is the second most important problem faced by secure system developers. Access control vulnerability is where an attack culminates: an attacker compromises a system using an unvalidated input or a bad authentication vulnerability, but eventually profits from breaking access control and accessing unauthorized information.

This chapter describes 7 security-oriented program transformations. These program transformations add authorization components, limit privileges during process creation, and use cryptography to protect confidentiality and integrity of data.
Chapter 10

Security-oriented Program Transforms for Proper Error Handling

In a broad sense, all security vulnerabilities originate from an application’s failure to handle error state. For example, a system failing to handle a malicious input goes to an erroneous state during an injection attack. However, these security vulnerability classes have been described in their specific chapters. The focus of this chapter is on how to handle error after it has occurred, typically by generating error messages and storing information for forensics.

Error messages serve various purposes. At design and development time, they can provide valuable debugging information. Error messages in an application can guide a user to the proper usage method. These are all useful. However, error messages can reveal internal information to an attacker, who can use it to launch another attack.

Another important task related to error handling is periodically logging system state information. This is useful for forensics: finding the actual cause of an error. In many network applications, logs are automatically monitored for intrusion detection. Sometimes, the system state information can be used to restart an application after it has crashed.

This chapter describes 5 security-oriented program transformations that allow system errors to be handled more efficiently. Some handle how error messages are presented to users. Others introduce components to periodically log system states.
10.1 Improper Error Handling Vulnerability

This chapter focuses on 3 error handling issues faced by developers.

The first is to determine how an error is handled in a system. An application with insufficient error handling crashes frequently. All error states should have explicit error handling routines, or at least a catch-all error handling routine that will allow an application to run beyond an error state. At the same time, error messages that describe the error state to its user should be carefully designed. Applications that produce overly verbose error messages, reveal internal details to an attacker. Attackers use this weakness for launching reconnaissance attacks. If a system shows internal error messages such as stack traces, database dumps, and error codes, attackers learn the implementation details that should never be revealed. An example is passing a malformed SQL query to get information about the database schema, popularly known as the blind sql injection attack.

Most attackers, especially those exploiting unvalidated input vulnerability, probe an application before launching an actual attack. Hence, improper error handling vulnerability has a significant impact on how vulnerable the overall application is, a fact that is not revealed by counting the vulnerabilities only. An unvalidated input vulnerability may as well be the end result of an improper error handling vulnerability, although vulnerability lists such as Bugtraq lists it under the unvalidated input vulnerability category. We have surveyed the Bugtraq vulnerability list [34] on several random weeks, and found that on average 6% of the attacks are related to improper error control vulnerability. This low percentage is not representative of the reality.

The second issue is to handle an application crash. When an application crashes, it should restart gracefully from it. Restarting gracefully means an application should restart to a state as close as possible to the state it was running before the crash. It does not lose data between system crashes.

The third issue faced by developers is to identify the root cause of any attack. Applications usually log important information to keep a dynamic trace, which helps a developer detect the root cause behind a failure. Application developers need to introduce logging mechanism to an application; they also have to ensure the confidentiality and integrity of logged data.
10.2 Solution Strategies

The 5 program transformations described in this chapter follow three solution strategies to combat improper error handling attacks.

1. Add components that store state information for auditing and logging.
2. Store system state periodically so that a system can restart gracefully.
3. Introduce exception handling in an application and mutate error messages so that they do not reveal internal information.

10.3 Adding Components for Storing State Information

10.3.1 Add Audit Interceptor Transformation

Auditing is an essential part of any security design. But audit requirements of an application change over the course of application development and use. You want to make it easy to add and change auditing events.

Add a separate auditing component that intercepts and audits event in the business tier of an application.

Motivation

The goal of a security audit is to ensure that events and actions in an application occur according to specified policies. Several aspects of an application can be audited, e.g. application development process, coding standards, runtime activity etc. The focus of this program transformation is to create an auditing component that will log audit events. Typically, the business tier of an application is the prime target for auditing, since it contains the bulk of business processing logic. An audit log records events, which is used for forensic purposes.

Preconditions

A program with the following characteristics benefits from an Add Audit Interceptor transformation

1. The program goes through a security audit.
2. Exactly what is to be audited is not fixed: the auditing requirements change as organizational policies evolve.

**Mechanism**

A developer specifies where to intercept data to audit, and how to create audit data.

The transformation adds an audit interceptor component [194] that intercepts requests and responses, and creates audit events. It first creates a Façade [74] of the audit points. At the façade, the incoming request is delegated to an auditing component before it is sent to the target. The auditing component intercepts all the information, generates audit data from the input data, and stores it.

**Example**

Suppose Alice is a Java developer. She wants to introduce an auditing component at the business layer of her application. In order to apply the transformation, she identifies the methods that need to be audited, suppose she selects a method called `foo`. The Add Audit Interceptor transformation creates a Façade [74] for this method in a separate Session Façade [5] layer. The `foo` function in this layer delegates the incoming request parameters to the audit component, also created by the transformation. After that, it forwards the request to the target `foo` and `bar` function. Figure 10.1 shows how audit data is processed.

![Figure 10.1: Processing audit data with an audit interceptor component](image)

The audit component maintains an event catalog, i.e. a mapping of requests to audit events. The event catalog is also specified by Alice. Finally, audit data is stored in an audit log.
Towards an Add Audit Interceptor Tool

An Add Audit Interceptor tool makes three changes: 1) it creates a Façade [74] for the methods to be audited and concentrate them in a Session Façade [5] layer, 2) it creates an audit interceptor component by composing classes and delegates requests to this component, and 3) it creates an event catalog from user specification that specifies how audit information is harvested. All of these tasks can be automated.

An alternative to creating a Façade [74] is to add the audit interceptor component as an Observer [74] at the entry point of the business logic layer. Another implementation strategy is to use aspect oriented programming: consolidate auditing code to a separate aspect. Aspects can be inserted into the methods either during the build process or through post-compile bytecode insertion. Aspects make the step of creating a Session Façade [5] layer unnecessary, but it may introduce performance penalties depending on the implementation.

An audit interceptor component should be scalable to handle high volume of audit requests. A strategy is to dump the messages to a persistent message queue; the message queue will process the messages asynchronously and persist audit log. An example of this is to use a JMS queue for processing audit logs.

10.3.2 Secure Logger Transformation

Application state should be periodically logged for debugging and forensic purposes. You want to design a system that logs messages correctly and securely and in a timely manner.

Use a centrally controlled logging functionality that can be used in various places of an application.

Motivation

Log files keep a trace of system events and user activity. They are a lucrative target for attackers; by altering a log file, an attacker can hide his or her traces. A system administrator cannot detect an attacker’s presence without a log trail. Thus an attacker can break into a system multiple times.

Logging functionalities in a system should be centralized to avoid redundancy. Additional care should be taken to protect the confidentiality and integrity of log files.
Preconditions

A program with the following characteristics benefits from a Secure Logger transformation

1. The program needs to log sensitive information and protect the log from unauthorized modification.

2. Logging functionality is required in many parts of a program, but duplicating the logging code in multiple places is not an option.

3. The application programmer wants to apply cryptography for maintaining confidentiality and integrity. A programmer knows the cryptographic keys beforehand.

Mechanism

A developer specifies the messages to log, and policies to retain confidentiality and integrity.

The transformation adds a logging component that encrypts and signs logged data. The messages are delegated to the logging component. The component centralizes logging functionality.

The logging component secures the data it receives by applying two cryptographic methods. It encrypts the data with a symmetric key to protect confidentiality. It also calculates the message digest and signs the data to protect integrity. The modified data is written to the log.

Example

Suppose Alice is a Java developer. She wants to apply a Secure Logger transformation to create a centralized logging component. Alice selects the messages that are to be logged and applies the transformation. The program transformation creates a logging component that writes messages to a log. It delegates all logging messages to the logging component. The logging component also secures a logged message by encrypting it, calculating a message digest, and signing it. These operations are applied to the original message similar using a combination of Decorated Filter transformation (see section 7.4.1) and Encryption/Decryption transformation (see section 9.5.1) and Signature Generation transformation (see section 9.5.3). Figure 10.2 shows how a log message is encrypted and signed.

Alice specifies the keys to be used for encryption, message digest creation, and signature generation. There can be a single set of keys used for the entire log file; alternatively, there can be multiple set of keys.
Alice also specifies the cryptographic algorithms, and the corresponding filters are used. These filters extend the `EncryptionDecorator` and `SignatureDecorator` classes, not shown in figure 10.2.

**Towards a Secure Logger Tool**

An *Add Audit Interceptor* creates a centralized mechanism for logging events. A *Secure Logger* transformation replaces an audit interceptor component with one that provides data security.

A *Secure Logger* transformation tool composes classes to create a logging component, delegates logging messages to the component, and modifies messages cryptographically to secure them. These tasks can be automated.

There can be two strategies to implement a secure log [194]. The strategy described in the example is to secure each message cryptographically and store them in a log. An alternative implementation strategy will be to secure the log as a whole.

If the logging component is located remotely, the strategy to encrypt and sign messages is a better option. If the other option is used, the messages should be transmitted to the remote logging component in a secure manner.

The secure logger transformation protects confidentiality and integrity of data. Other strategies can be applied to further secure a log. A possible strategy for an attacker is to delete log entries. To protect log messages, they should be stored with a sequence number, that is calculated each time a new log message is entered. Thus, deleted messages in the log will have a break in the sequence number, and they can be subsequently detected.
10.4 Restarting Gracefully from Error

10.4.1 Checkpointed System Transformation

A component failure can result in loss or corruption of state information maintained by the failed component. You want to design a system so that its state can be recovered and restored to a known valid state in case a component fails.

*Store persistent state information all the time. Use a wide variety of configurations that provide the ability to restart the system from a known valid state (i.e. the checkpoint).*

Motivation

Application state is stored in memory. But memory does not survive a system crash. Hence, applications should periodically store state information. This stored information may allow an application to return to its state before a crash.

Consider a mail transfer agent such as Postfix. Postfix stores its mail queue in several directories in the file system. However, it keeps a copy of some of the mail queues in memory for performance. It periodically updates the file system, i.e. update the state of the mail queue. The disk-based queue structure provides persistent storage for mail messages. The state of mail delivery for each mail can be identified by the subdirectory it is currently in. Even if Postfix crashes, the storage of data and state ensures that it can restart gracefully. In the worst case, some mails can be delivered multiple times, but mails are not lost.

Preconditions

A program with the following characteristics benefits from a Checkpointed System transformation

1. The program may crash frequently.
2. The program stores the internal state in memory. It is possible to exchange the state information between memory and disk.

Mechanism

A developer should specify the information to persist and checkpoints in the system.
The transformation introduces the method calls to save information. Typically, it creates a storage component and passes it the state information.

Example

Suppose Alice is a Java developer. She wants to store system state applying a Checkpointed System transformation. The transformation creates a component that writes to disk files. Alice identifies the state information to pass, and where to store the information (example of state information may be a shopping cart, a document that is edited but not saved etc). She also identifies the places in the program where information is stored. The transformation introduces method calls and delegates the information to the storage component.

Towards a Checkpointed System Tool

A Checkpointed System tool makes two changes: 1) it composes a storage component, and 2) it introduces method calls in the program where data needs to be stored.

An alternative to implement this is to use aspect oriented programming: use and around advice similar to an Add Audit Interceptor transformation. Aspects can be inserted into the methods either during the build process or through post-compile bytecode insertion.

10.5 Handling Error Messages

10.5.1 Error Message Suppressor Transformation

You want to ensure that error messages do not reveal internal information.

Suppress error messages by applying suppression policies and generate a sanitized error message that does not reveal internal information.

Motivation

Verbose error messages reveal internal information to an attacker. The attacker can use this knowledge to launch an attack. Suppose an attacker wants to launch an SQL injection attack to an application. He can generate cleverly fashioned SQL statements that will lead to failure, feed them to the application, and
check the error messages. Some applications provide detailed information about where the SQL statement failed, and include stack traces, database table names, data types, etc. The attacker now knows these internal information. He or she can then formulate the SQL injection attack vector.

**Preconditions**

A program with the following characteristics benefits from a *Error Message Suppressor* transformation

1. The program reveals detailed information about system internals.

2. The suppression policy is not uniform, hence a global policy cannot be used. Error messages that are stored in the log should not be suppressed. Only error messages provided as a feedback to the user should be suppressed.

**Mechanism**

A developer specifies the interception point for error messages (e.g. intercepting all write messages to `stderr`), and the policies applied to suppress error messages.

The program transformation adds an interceptor component that applies the suppression policies. The suppression policies are added by applying a *Decorated Filter* transformation (see section 7.4.1) on error message strings.

**Example**

Suppose Alice is a Java developer. She wants to apply an *Error Message Suppressor* transformation on an error message. Alice selects an error message in a catch block that prints the entire stack trace.

```java
... try{
 ... } catch (AnyException e) {
 e.printStackTrace();
 } e.printStackTrace();
 ...
```

She wants to apply two suppression policies. The first policy removes all line numbers from error messages, and the second policy removes the entire stack trace. Figure 10.3 how a *Decorated Filter* transformation (see section 7.4.1) adds the suppression policies.
Towards an Error Message Supressor Tool

An Error Message Supressor tool applies a Decorated Filter transformation (section 7.4.1) on error message strings. Another mechanism is using aspects by adding a before advice.

A developer has to specify which output variables to intercept and the error suppression policy. A tool can contain a library of policies from which a developer selects the appropriate ones for a context. The policy library should be extensible so that developers can add their organization-specific custom policies. One example policy is to group multiple error messages and provide a generic error message. Consider, two error cases in an access control context—1) a request for a non-existent file and 2) a request for an existing file for which a user does not have access permissions. Typically, separate error messages are generated for separate error cases. Unifying the error messages to a generic message will not reveal to a requester whether a file exists in a system or not.
10.5.2 Exception Shielding Transformation

You have applied rectification policies to input variables to prevent input validation errors. You want to preserve application behavior when rectified user inputs cause an unexpected state. Also you want to shield internal information from external users.

*Use an exception when rectified data is used. Apply policy to exception output so that they do not reveal internal information.*

Motivation

A *Decorated Filter* transformation (see section 7.4.1) often modifies input in order to validate it. For example, a filter that removes SQL keywords from an input can eventually produce an SQL statement that is not correct. In this case, the application does not have an SQL injection vulnerability, but it may crash when it attempts to process an incorrect SQL statement.

A security-oriented program transformation preserves good path behavior. The filters applied to a non-malicious user’s input should not modify it so that it leads to an invalid state. Only the rectified attacker inputs can take an application to an invalid state. Every time such a modification is done, an exception should be thrown to indicate a potentially threatening input.

Not all modifications indicate an attacker’s input. Some modifications canonicalize data inputs, others apply encryption, digital signature etc. It is valid for these program transformations to modify user input.

Preconditions

A program with the following characteristics benefits from a *Exception Shielding* transformation

1. An application developer has applied a *Decorated Filter* transformation (see section 7.4.1) that modifies the input. The modified input can introduce an unexpected state.

2. Information related to exceptions should be returned to the user. However, exception details contain clues that an attacker can use to launch another attack.
Mechanism

A developer specifies exception type and insertion point and marks the filters that potentially rectify input.

The program transformation defines new exceptions. It modifies the filters to throw an exception when they rectify an input. At the top level it catches the exception and obfuscates the error message shown to the user. Policies are applied to make safe exceptions that do not contain sensitive information in the exception message, and do not contain a detailed stack trace. The latter part is added by an Add Error Message Suppressor transformation (see section 10.5.1).

Example

Suppose Alice is a Java developer. She has applied a Decorated Filter transformation to an application to rectify inputs to prevent SQL injection attacks. The Decorated Filter transformation has generated the following code (copied from the example section of Decorated Filter transformation in section 7.4.1).

```java
... public class DBConnect {
  ...
  public void showData() {
    ...
  UnsafeString username = new UnsafeString();
  try {
    username.setStr(stdin.readLine());
  } catch (IOException e1) {

  try {
  stmt = connection.createStatement();
  PolicyDecorator policy = new SQLPolicyRemoveAnd(new SQLPolicyRemoveOr(username));
  resultSet = stmt.executeQuery("select * from users where username = "+
  policy.convert().getStr() + ");
  } catch (SQLException e) {
  e.printStackTrace();
  }
  }
}

Alice wants to apply the Exception Shielding transformation. She identifies the filters applied to the variable (SQLPolicyRemoveAnd and SQLPolicyRemoveOr in this example), and indicates that both of these policies modify a malicious input. The program transformation creates a new exception hierarchy (see figure 10.4) and modifies the filter classes so that they throw an exception when they modify input.

The DecoratedFilterException is the superclass of the newly defined exceptions. Filters that do not modify any input throw NotModifiedException. ModifiedException is thrown by filters
that modify both valid and invalid inputs, e.g. canonicalize them, encrypt or decrypt etc. Filters that modify malicious inputs throw MaliciousModifiedException. At the highest level, the try-catch block will catch MaliciousModifiedException. It will then apply Error Message Suppressor transformation (see section 10.5.1) to obfuscate error messages. In this example, Alice selects two suppression policies: SuppressionPolicyRemoveLineNumber and SuppressionPolicyRemoveStackTrace.

The final output is shown here,

```java
... public class DBConnect {
    ...
    public void showData() {
        ...
        try {
            stmt = connection.createStatement();
            PolicyDecorator policy = new SQLPolicyRemoveAnd(new SQLPolicyRemoveOr(username));
            resultSet = stmt.executeQuery("select * from users " + "where username = "+
                policy.convert().getStr() + "]");
        } catch (MaliciousModifiedException e) {
            UnsafeString errMsg = new UnsafeString (e.getMessage());
            SuppressionPolicyDecorator suppressionPolicy =
                new SuppressionPolicyRemoveLineNumber (new SuppressionPolicyRemoveStackTrace (errMsg));
            System.out.println (suppressionPolicy.convert().getStr());
        } catch (SQLException e) {
            ...
        }
    }
}
```

Note that, each of the policy classes are also modified so that their convert methods throw various exceptions. The following pseudocode describes how the exceptions are incorporated.
public class SQLPolicyRemoveAnd extends PolicyDecorator {

    public AbstractStringContainer convert() throws Exception {
        // Check whether a MaliciousModifiedException exception has already occurred
        try {
            component = super.convert();
        } catch (NotModifiedException e) {
        } catch (MaliciousModifiedException e) {
            throw e;
        }
        ...
        // Apply policies to input
        ...
        // Check whether input is modified or not and throw exceptions
        if (initial and modified input are the same)
            return input
        else
            if (the modification only affects malicious input)
                throw new MaliciousModifiedException;
            else
                return input;
    }

Towards an Exception Shielding Tool

An Exception Shielding tool creates a new exception hierarchy, modified filter classes of a Decorated Filter transformation (see section 7.4.1), and applies Error Message Suppressor transformation (see section 10.5.1). All of these steps can be automated.

10.6 Organizing the Transformations

The program transformations are organized using two criteria: their impact on codebase and the type of transformation. Table 10.1 describes how they are organized.

Program transformations to fix improper error handling are small. They typically mutate error messages to suppress internal information. These program transformations occur frequently in a system.

<table>
<thead>
<tr>
<th>Change</th>
<th>Small, infrequent change</th>
<th>Small, frequent change</th>
<th>Large change</th>
<th>Change beyond boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Component</td>
<td>Add Audit Interceptor</td>
<td>Checkpointed System</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Secure Logger</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutate data</td>
<td>Error Message Suppressor</td>
<td>Exception Shielding</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1: Security-oriented Program Transformations for Improper Error Handling

The program transformations described in this chapter are of two types. There are program transformations that add a logging component. Others mutate error messages to hide internal information.
Add component. Adding components for keeping track of internal error messages and internal states during execution indirectly help in error handling. Adding a simple logging component requires a small change. However, features such as message digest, signature, encryption can be added for producing more secure logs. Eventually a logging component can be quite complex and large if these additional transformations are applied on a logging component.

Mutate data. Applications frequently generate error notifications, e.g. out of memory, null pointer exceptions, system call failure, database unavailable, network timeout, malformed input, etc. An Error Message Suppressor transformation is similar to a Decorated Filter transformation, because both decorate [74] a variable with various policies; only difference is that the policies applied by an Error Message Suppressor are for suppressing error messages. An Exception Shielding transformation also uses the Error Message Suppressor transformation.

10.7 Composing the Transformations

A Secure Logging transformation combines many other transformations: Decorated Filter (see section 7.4.1), Encryption/Decryption (see section 9.5.1), Message Digest Creation (see section 9.5.2), Signature Generation (see section 9.5.3), etc.

An Exception Shielding transformation follows a Decorated Filter transformation (see section 7.4.1). The Exception Shielding transformation combines Error Message Suppressor transformation for output obfuscation. Applying these three transformations in any other order does not make sense.

10.8 Summary

Improper error handling vulnerability is often under-rated since the vulnerability databases do not directly list instances of these. However, many other vulnerabilities can be exploited because an attacker gains knowledge about application internals using this vulnerability.

This chapter describes five security-oriented program transformations. These program transformations handle how error situations are handled, how error messages are created and how system events are logged.
A denial-of-service (DoS) attack is an attempt to prevent legitimate users from accessing information or services. The most common target of a DoS attack is a computer and its network connection, typically by flooding it with garbage requests. Another variant of DoS attack is spamming a user. The most common strategy for preventing these attacks is to use a firewall to filter incoming request, e.g. using an ingress or egress filter at network intermediaries, using a black list to filter spam-friendly domains etc. A firewall is an external, off-the-shelf component that is introduced in between an application and the network. Firewalls and firewall-based prevention of network denial of service attacks are not the focus of this chapter.

This chapter focuses on denial of service attacks on a software application and their prevention. These strategies follow two extreme trends. The first trend is to be parsimonious and use resources sparingly inside an application, and check the availability of a resource before using it. The other extreme is to add redundancy so that a redundant component may take over if another component fails. This chapter describes 5 security-oriented program transformations to prevent application level denial of service.

### 11.1 Denial of Service Vulnerability

The target of all application level denial of service attacks is to consume some required resource, and make it unavailable to the application. Once an attacker consumes the resource, another user using the application is denied of that resource. Examples of scarce resources include bandwidth, database connections, disk storage, CPU, memory, threads, etc. Consider a mail transfer agent, that spawns a new process to handle every incoming email. An attacker can send a lot of garbage emails to eventually make the application reach its maximum process limit, and therefore deny legitimate users from connecting to the MTA.

Other variants of denial of service attacks target system resources related to a particular user. An example is locking out a user by sending invalid credentials to an application’s authentication component.
Another type of application level denial of service attack is to ‘capture and hold’ shared resources. For example, an attacker can acquire lock on a shared resource and deny other users from accessing the resource.

Finally, some attacks are targeted to make the whole application unavailable. Even an unsuccessful attempt of injection attack can crash an application. The target of a denial of service attacker can be to feed an application with spurious data. This serves two purposes. First, the application spends a lot of time validating input and may fail to serve legitimate requests. Second, the application may crash and become unavailable.

11.2 Solution Strategies

The most common strategy for preventing denial of service attacks is to add ingress and egress filters in network intermediaries. But distinguishing attack traffic from ordinary traffic is not a trivial task. Since IP address spoofing is very common, the origin of a packet can be spoofed; therefore, attackers can trick network filters into accepting spurious traffic. Filtering reduces traffic in many cases, but it is not the most effective solution for preventing denial of service attacks. One program transformation in this chapter applies a filtering policy: it introduces a component for caching messages and uses it to identify and discard replayed messages.

Instead, a defensive mechanism is to balance load into multiple redundant applications. Redundancy is applicable in preventing application level denial of service as well. Among the 5 security-oriented program transformations in this chapter, 3 transformations describe how redundant components can be created and applied for load balancing.

Opposite of redundancy is parsimonious use of resources. One program transformation promotes this strategy by imposing resource limits and introducing checks.

11.3 Adding Components for Limiting Incoming Requests

11.3.1 Message Caching Transformation

You want to protect your system from replay attacks.

*Cache incoming messages; match all new messages with the cache, and reject if a message is replayed.*
Motivation

In a replay attack, a valid data transmission is maliciously repeated. An application’s failure to detect stale and replayed data opens up a denial of service opportunity, but the attacks can be more severe. Suppose an application assigns a cookie to an authenticated user, and then uses the cookie to re-authenticate. An attacker can intercept the cookie and apply it to bypass the authentication step. The authentication vulnerability in this example originates more from an architectural mistake (not re-authenticating each time), but nevertheless, it shows how severe vulnerabilities can originate from failure to identify replay attacks.

A way to avoid replay attacks is using unique tokens for each session, so that each of the messages is different. Various strategies can be applied to distinguish messages: using sequence number, using nonces with Message authentication code (MAC), using timestamps etc.

An application can cache messages and identify replayed messages when an incoming message is already in cache. The check for a cache hit should be performed as early as possible, ideally in a policy enforcement point component at the system perimeter.

Preconditions

A program with the following characteristics benefits from a message caching transformation

1. The program has a limited number of access points and policy enforcement points where an authentication component is introduced. A developer may apply Single Access Point (section 6.2.1) and Policy Enforcement Point (section 6.2.2) to create policy enforcement points.

2. The program communicates using a protocol that uses unique tokens for each session.

Mechanism

A developer specifies the program perimeters that accept incoming messages. A developer also specifies how to uniquely identify incoming messages.

The transformation implements a message cache with replay detection mechanism. All incoming messages are checked. The messages stored in the cache have a unique identifier. All stale messages are dropped by the replay detection mechanism.
Example

Suppose Alice is a Java developer. She wants to apply a *Message Caching* transformation to a program. The program communicates using a protocol that uses unique sequence number to identify each message. Alice chooses the policy enforcement point of the application and identifies the part of the message that contains a seq number. The program transformation composes classes to create a cache, that distinguishes incoming messages by calculating the hash of the sequence number. If there is a cache hit, the message is dropped.

Towards a Message Caching Tool

A *Message Caching* transformation composes classes to create a cache component at the policy enforcement point. The part that varies is how to identify unique messages. If a protocol uses unique session tokens, then these can be used to find replayed messages. However, session tokens can be forged. Typically, protocols that use session tokens also include a signed hash to verify the origin of the message. In such a case, the cache component will also have to verify the signature; perhaps a *Signature Generation* (see section 9.5.3) transformations is applied to include this feature. In such a case, a user has to specify more things, e.g. the key used to sign etc. Cryptographic operations, especially public key cryptography, are expensive; hence this may lead to denial of service attempts by feeding the caching component spurious messages with fake keys. The use of verification in a caching component is therefore a design decision that depends on typical message load, use of cryptographic keys etc.

11.4 Using Resources Parsimoniously

11.4.1 Resource Management Transformation

Denial of Service is tackled by adopting several network-based strategies. However, system architecture should be resilient to such attacks as well. You want to design a system that manages available resources so that an attacker cannot launch an application level denial of service attack.

*Manage resources such that every resource is freed up when it is not in use. Explicitly check whether a resource is available before using it.*
Motivation

Application resources are limited. One way of launching a denial of service attack is to exhaust the limited resource. A typical problem is that a resource is consumed, but not freed up upon use. Examples of such problems are memory allocated but not freed, socket opened but not closed, database connection opened but not closed etc. Resource management problems can give an attacker an opportunity to launch a denial of service attack.

Programs also crash when they attempt to access a resource that has not been allocated. For example, a null pointer exception in Java denotes that a null object is referenced. Programmers are supposed to test for null-ness before using an object, but mistakes are very common.

Another common resource management mechanism is to adopt operating system’s resource management features. For example, one can set resource limit for running processes. This can be done by using per-process resource management or by adopting operating system’s resource management features, e.g. maxproc rlimit sets the number of processes per UID.

Preconditions

A program with the following characteristics benefits from a Resource Management transformation

1. Program uses resources that go through allocate-use-free cycle. Programmers fail to free resources that have been allocated.

2. Programmers access resources without checking whether they are available.

Mechanism

A developer identifies the program variables that represent resources, and the management policy, e.g. free after use, or check before access etc.

The transformation adds resource management checks at the program points.

Example

We will describe two examples of Resource Management program transformation. The first subject of program transformation is a Java program that improperly manages an HTTP connection. The second
subject is a C program that has segmentation fault because it attempts to reference null objects.

**Failure to Close an HTTP Connection.** The following Java method downloads a text file from server.

```java
private void downloadPage(String TextFileName) {
    InputStream is = null;
    HttpConnection conn = null;
    try {
        String url = "http://TomcatserverIP:8080/Folder/textfile.txt";
        conn = (HttpConnection) Connector.open(url);
        is = conn.openInputStream();
        InputStreamReader r = new InputStreamReader(is);
        char[] buffer = new char[32];
        StringBuffer sb = new StringBuffer();
        int count;
        while ((count = r.read(buffer, 0, buffer.length)) > -1) {
            sb.append(buffer, 0, count);
        }
        str = sb.toString();
    } catch (IOException ex) {
        ex.printStackTrace();
    } finally {
        try {
            is.close();
            if (conn != null) {
                conn.close();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Despite careful check, the program may not close the `HttpConnection` variable in line 23 if there is an exception in line 22. A programmer, with a Resource Management tool can select `is` and `conn` variable and apply the policy that a resource should be freed after use. The tool will move the part that closes the connection inside a finally block.

```java
... 
      } finally {
          try {
              is.close();
          } catch (IOException ex) {
              ex.printStackTrace();
          }
      }
  }
... 
```

188
This will ensure that the connection will be closed in all cases.

**Check Before Access.** The following is a C function that displays the content of a tree. It can have a segmentation fault if the pointer ptr in lines 4, 7 and 9 are null.

```c
... void disptree(node * ptr)
{
    if ((ptr->left) != NULL)
        disptree(ptr->left);
    cout<<"Position:"<<ptr->pos<<" Data:"<<ptr->data<<endl;
    if ((ptr->right) != NULL)
        disptree(ptr->right);
}
```

A programmer can apply a *Resource Management* transformation on the `ptr` variable. The program transformation will introduce a nullness check before all accesses to `ptr`.

```c
... void disptree(node * ptr)
{
    if ((ptr != NULL) && ((ptr->left) != NULL))
        disptree(ptr->left);
    if (ptr != NULL) {
        cout<<"Position:"<<ptr->pos<<" Data:"<<ptr->data<<endl;
        if ((ptr != NULL) && ((ptr->right) != NULL))
            disptree(ptr->right);
    }
}
```

**Towards a Resource Management Tool**

There are many operating system directives for resource management. Disk space is managed by partitioning and limiting the space with quotas. File size can be limited by OS directives (like `rlimit`). Similar ideas apply for program memory segments, e.g. data segment and stack segment. A program transformation tool can use those directives.

In order to test for resource leakage a program transformation tool can use various trace analysis tools, e.g. valgrind for memory leaks. Similarly, there are various tools for checking null-ness of objects. There are some recent tools that use pluggable type systems for detecting null objects, e.g. JustAdd [64] [66],
JavaCOP [128] and the Checker framework [154]. These tools can be used to analyze a program to find appropriate transformations.

One of the hardest parts of denial of service attacks is determining which part of an application is vulnerable. Load testing tools, such as JMeter can generate web traffic so that one can test how a site performs under heavy load, e.g. how many requests per second an application can process. To determine if any resource can be used to create a denial of service, a developer should analyze each one to see if there is a way to exhaust it. This analysis guides the subsequent program transformation.

11.5 Throwing in Redundancy

11.5.1 Replicated Component Transformation

Components of an application fail for many reasons. This hinders the natural workflow, especially in transactional applications. You want to assure availability of application components in the midst of such failures.

Replicate components and during failure replace the failed component with an available one. Perform load-balancing based on some workload scheduling mechanism by a workload management proxy.

Distribute the replicated components at multiple points in the network.

Motivation

Usually when an application component crashes, it takes the whole application down. An attacker can exploit this vulnerability by continuously crashing an application and deny other users from its service. One method to prevent a component from crashing an application is to decouple it and distribute it. Furthermore, application components are replicated on separate systems. It can be data replication if the same data is stored on multiple storage devices, or computation replication if the same computing task is executed many times. Replication ensures consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or availability.

Closely related to replication is the issue of load balancing. Replicated components should be controlled by a load balancing component. A load balancing component distributes a load of different computations
across machines. Additionally, it monitors the replicas and allows a single computation to be dropped in case of failure.

**Preconditions**

A program with the following characteristics benefits from an *Replicated Component* transformation

1. The program is a transactional system, but the components are faulty. As well as component failures, link failures are common.

2. Faults in one component are not expected to be strongly correlated with similar or identical faults in another component. Faults are caused by environmental factors external to component, not design or implementation errors.

3. It is possible to create redundant components and distribute them to separate network locations. Each distributed component can be stateless or stateful, but the components do not share state.

4. Each distributed component can reliably access the overall application’s state.

5. The API of the components cannot change.

**Mechanism**

A developer identifies the components to replicate, the network points where replication is required and describes the load balancing mechanism in workload balancing policy.

The transformation creates replicas at the network points. It also introduces a workload management proxy that monitors the replicas and switches to another component when one component fails.

The replications are non-transparent to an external user. The replicas and the workload management component are example of the Proxy [74] pattern.

**Example**

Suppose Alice is a Java developer. She wants to apply a *Replicated Component* transformation on a Java application. She chooses the component to replicate. In this example the component that contains function `foo` is to be replicated. She also identifies two remote locations that will host the replicated components
and selects a load balancing algorithm that will be used by the workload management component. The program transformation creates the replicas and workload management component as Proxies [74]. Figure 11.1 shows the new classes that are introduced.

![Figure 11.1: Classes introduced by a Replicated Component transformation](image)

The whole replication is transparent to the end user. After the transformation, a client’s requests are directed to the workload management proxy component instead of the original component (both have the same API). The component selects the best replica according to a load balancing algorithm and delegates the request to it. It also monitors the execution and redirects requests in the case of failure.

**Towards a Replicated Component Tool**

The hardest part for a developer is to select the component and functionality to replicate. Often, it is not easy for a developer to identify a particular component. In that case, existing refactoring tools can be helpful. Suppose, one particular method `foo` of a class is to be replicated, but not another method `bar`. In that case, a developer can apply refactorings [71] such as *Move Method*, *Split Class* etc to isolate the component to be replicated.

Once a developer identifies the component, a program transformation tool will create Proxies [74]; this step can be automated. Then the replicas are distributed among the identified network points. In this step, local calls are replaced with remote calls; a similar strategy is applied in the *Partitioning* transformation (see section 6.1.1) too.

A *Replicated Component* tool creates a workload management proxy component that implements a load balancing and a monitoring mechanism. A developer has to choose the appropriate one from a set
of available mechanisms. A useful tool should present a lot of choices for load balancing and monitoring mechanism. Moreover, the tool should be extensible, so that developers can add their own custom mechanisms.

A Replicated Component transformation creates replicas that can be stateless of stateful, but they do not share state. If the replicas are stateful and must be kept consistent, more components should be introduced to ensure consistency of state across components.

11.5.2 Standby Transformation

Every system component is exposed to failure. You want to design a system that is tolerant of single component failure, i.e. the failure of a single component will not cause a system outage. Additionally, you want the replicas to share state information.

Structure a system with backup components so that the service provided by one component can be resumed by the backup component in case of system failure. The original component passes state information to the backup component; the backup component uses state information to resume service.

Motivation

The fundamental assumption to reliable application design is that there will be failures. Often the failures have nothing to do with error in design and implementation, but they are caused by environmental factors. An attacker can exploit this vulnerability by continuously crashing an application and deny other users from its service. Replication is a common method to protect a component from denial of service attacks. Replicated components operate independently, they do not share system state. When components need to share state, state storage and distribution mechanism needs to be added with replicated component.

A standby or a hot spare is a failover mechanism to provide reliability in system configurations. The hot spare is active and connected as part of a working system. When a key component fails, the hot spare is switched into operation. Having a standby reduces the otherwise significant start-up delay.

At one extreme, a complete application can be duplicated and kept up-to-date. More often, a standby is a single vital component without which the entire application will fail. A management component controls the original component and standby; it decides when to use the spare component.
A standby does not necessarily give 100% availability or protect against temporary loss of the system during the switching process; it is designed to significantly reduce the time that the system is unavailable.

**Preconditions**

A program with the following characteristics benefits from a *Standby* transformation:

1. The program is a transactional system, but the components are faulty. As well as component failures, link failures are common.

2. Faults in one component are not expected to be strongly correlated with similar or identical faults in another component. Faults are caused by environmental factors external to component, not design or implementation errors.

3. It is possible to create redundant components and distribute them to separate network locations.

4. It is feasible to externalize component state information. The distributed components can share state information.

5. A small number of transactions occurring between the time a component fails and the time service is restored using a backup component are irrelevant or inconsequential, or can be recovered and reapplied. In other words, 100% availability is not necessary.

6. The functionality that is distributed is such that it can be resumed by using state information.

7. The API of the components cannot change.

**Mechanism**

A developer identifies the component to replicate or create a backup. He or she also specifies the network points where replication is required. Finally, a developer points out what state information of the original component are to be kept.

The transformation creates replicas at the network points. Typically there is one replica: the backup component. The transformation also introduces a proxy that monitors the original component, manages state information and passes computation and state to the standby when the original component fails.
The replications are non-transparent to an external user. The backup and the management component are example of the Proxy [74] pattern. The state information is stored following the Memento [74] pattern.

**Example**

Suppose Alice is a Java developer. She wants to apply a Standby transformation on a Java application for photographic post-processing. The application takes a jpg file and applies several photographic filters on the jpg and saves it. Alice wants to create a standby component, so that a failing component will not lose the processing job in hand; instead, the processing is switched to a standby component, along with the saved state. Alice chooses the addFilter and removeFilter functions of the PostProcessingComponent class and applies the Standby transformation.The transformation creates a proxy that manages how the request is processed and creates an ActiveComponent as well as a StandbyComponent. The application state is stored in ActiveComponent. For the photo processing application, the state information contains the filters applied to a jpg image. In the event of a crash, RecoveryProxy passes it to StandbyComponent. Figure 11.2 shows the new classes that are introduced.

![Diagram](image-url)

**Figure 11.2: Classes introduced by a Standby transformation**

The whole replication is transparent to the end user. After the transformation, a client’s requests are directed to the proxy component instead of the original component (both have the same API). The component runs the computation in the original component, and redirects it to the standby in the case of failure.
Towards a Standby Tool

The hardest part for a developer is to choose the candidate for standby. Often, it is not easy for a developer to identify a particular component to replicate. In that case, existing refactorings can be applied to isolate the component to replicate.

Once a developer identifies the component, a program transformation tool will create Proxies [74]; this step can be automated. The application state is stored with a Memento [74]; the state storage functionality can be introduced automatically. Finally, the standby component is distributed to an identified network point. In this step, local calls are replaced with remote calls; a similar strategy is applied in the Partitioning transformation (see section 6.1.1) too.

A Standby tool creates a recovery proxy component that monitors the active component and decides when to switch to the standby component. A developer has to choose the appropriate monitoring mechanism from a set of available ones. A useful tool should present a lot of choices for monitoring mechanism. Moreover, the tool should be extensible, so that developers can add their own custom mechanisms.

11.5.3 Tandem Component Transformation

Failure in one portion often starts the domino-effect and the failure is propagated to bring about the entire system failure. You want to detect and isolate a failing component, so that an independent failure of one component will not cause a system failure.

*Replicate components at various network points. Run the same computation in all the replicated components, and use a component to compare the output or the internal states. Accept output only if the components agree.*

Motivation

Often a failed application component takes the whole application down. It is important to detect component faults quickly, or to detect component faults at a specific point during processing, to prevent component faults from causing system failures. At the same time, computation should continue despite attackers attempt to crash an application. One method to prevent a component from crashing an application is to decouple it and distribute it.
But replication does not allow a developer to quickly detect which part of the system failed. Inspection of the output of a single component may not directly reveal a fault. Instead, comparing the result produced by replicated components indicate whether a fault has occurred or not.

**Preconditions**

A program with the following characteristics benefits from a *Tandem Component* transformation

1. The program is a transactional system, but the components are faulty. As well as component failures, link failures are common.
2. Faults in one component are not expected to be strongly correlated with similar or identical faults in another component. Faults are caused by environmental factors external to component, not design or implementation errors.
3. It is feasible to externalize component state information. The distributed components can share state information.
4. It is feasible to compare the outputs or internal states of components.
5. Duplicating system components is economical.

**Mechanism**

A developer identifies the component to replicate. He or she also specifies the network points where replication is required. Finally, a developer points out what state information of the original component are to be kept and how state informations are handled by a comparator component.

The transformation creates replicas at the network points. It also creates comparator components to check results. A typical setup consists of an even number of replicated components (often four or more), organized as sets of pairs, together with a comparator for each pair.

The comparator component created by the transformation keeps state information of the replicas. It matches the state information produced by each member of its replicated pair. If the states do not match, the comparator concludes that a fault has occurred in one of the components and takes corrective action.

The replications are non-transparent to an external user. The comparator component is an example of the Proxy [74] pattern. The state information is stored following the *Memento* [74] pattern.
Example

Suppose Alice is a Java developer. She wants to apply a Tandem Component transformation on a Java application. She chooses the component to replicate. In this example the component that contains function \( \text{foo} \) is to be replicated. She also identifies two remote locations that will host the replicated components. The program transformation creates the replicas and comparator component. Figure 11.3 shows the new classes that are introduced.

The transformation creates a Comparator component and two tandem components as Proxies [74]. Both tandem components store their state in Memento-s and share them with the Comparator.

The whole replication is transparent to the end user. After the transformation, a client’s requests are directed to the Comparator component instead of the original component (both have the same API). The Comparator delegates the task to TandemComponent1 and TandemComponent2, matches their state and produces result.

Towards a Tandem Component Tool

Most of the tasks for a Tandem Component transformation can be automated. The manual specification is the choice of the candidate for replication. Often, it is not easy for a developer to identify a particular component to replicate. In that case, existing refactorings can be applied to isolate the component to replicate.
Once a developer identifies the component, a program transformation tool will create Proxies [74]; this step can be automated. The application state is stored with a Memento [74]; the state storage functionality can be introduced automatically. Finally, the tandem components are distributed to an identified network point. In this step, local calls are replaced with remote calls; a similar strategy is applied in the Partitioning transformation (see section 6.1.1) too.

11.6 Organizing the Transformations

Denial of Service in software is different from a denial of service attack on the network, e.g. a SYN flood attack. The primary goal of DoS attackers on software applications is to consume all of some required resource to prevent legitimate users from using the system. Another type of attack is to deliberately crash a component to make it unavailable. Strategies to survive these attacks vary; sometimes systems should be parsimonious and enforce a limit, while at other times they should act generous and use redundancy. The impact of the program transformations varies from small localized changes to large changes.

<table>
<thead>
<tr>
<th>Small, infrequent change</th>
<th>Small, frequent change</th>
<th>Large change</th>
<th>Change beyond boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Component</td>
<td>Message Caching</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Limit</td>
<td>Resource Management</td>
<td></td>
<td>Replicated Component</td>
</tr>
<tr>
<td>Distribute</td>
<td></td>
<td>Standby</td>
<td>Tandem Component</td>
</tr>
</tbody>
</table>

Table 11.1: Security-oriented Program Transformations for Denial of Service

The program transformations described in this chapter are of three types (see Table 11.1).

**Add component.** Attackers typically send multiple messages with garbage payload to overload CPU and memory. At the system entry point, a separate component can be added by an Message Caching transformation that intercepts incoming messages, looks for replayed messages, and drops them. However, the policies used to determine a replayed message should be very conservative, because it is difficult to distinguish a good traffic flood from a bad traffic flood.

**Limit.** Example of resources that are limited include bandwidth, database connections, disk storage, CPU, memory, threads, etc. Adding a resource limit check at multiple points of a program ensures that a program
does not crash or halt because of resource exhaustion. A tool can automatically add various types of resource limit checks. A tool will contain a library of policies from which a developer selects the appropriate ones, similar to other program transformations that apply policies (e.g. Decorated Filter in section 7.4.1).

**Distribute.** Redundancy is a common method to fight against DoS attacks that results in large changes. For example, a Replicated Component transformation results in duplicating the functionality of a component at multiple network points. The developer specifies which components to distribute, the number of copies, and their location. The transformation makes copies of system components and creates an additional load balancing component.

### 11.7 Composing the Transformations

Message Caching transformation is typically applied at the policy enforcement points. Hence, it can be added after Single Access Point (section 6.2.1) and Policy Enforcement Point (section 6.2.2) transformations.

Some strategies of a Resource Management transformation depends on the error handling scheme. These are typically used with various error handling transformations, e.g. Exception Shielding (section 10.5.2) and Error Message Suppressor (section 10.5.1).

A Standby transformation is dependent on the program state. It follows a Checkpointed System transformation (section 10.4.1) that keeps state information of the component. It creates replicas and adds a monitor that transfers control to a standby component when a component fails.

### 11.8 Summary

This chapter describes 5 security-oriented program transformations. These program transformations follow various strategies to prevent denial of service: some introduce checks to detect duplication and resource exhaustion, while other introduce redundant components to handle additional load.
Chapter 12

Defense In Depth with Program Transformations

Security-oriented program transformations are rarely used alone. Developers mitigate multiple security problems in every program they write; each of these security problems can be tackled with appropriate security-oriented program transformations. In some cases, a single program transformation can mitigate a type of vulnerability. But it is more common to apply a group of program transformations methodically to mitigate a vulnerability. The goal is to have defense in depth [210]: having layers of security protection in a program. Sometimes, the layers correspond to different vulnerabilities. Sometimes, the layers collaborate to mitigate a single type of vulnerability. In this chapter, we will describe two most common security vulnerabilities and how developers can apply program transformations to prevent an attacker from taking advantage of these vulnerabilities.

1. We will describe how multiple program transformations can be applied to prevent buffer overflow vulnerability in C programs.

2. We will describe how multiple program transformations can be applied to prevent SQL injection vulnerability in PHP programs.

12.1 Preventing Buffer Overflow Vulnerability in C Programs

Buffer overflow in C programs originate from 3 types of programming mistakes. We will describe these mistakes and program-transformation based preventive measures.
1. Developers use unsafe buffer handling functions. These functions do not check for buffer bounds, and can allow an attacker to write past the buffer bounds. A developer can apply Safe Library Replacement (section 7.4.3), and Safe Type Replacement (section 7.4.4) transformations to replace unsafe functions with safe library functions.

2. Developers use numeric datatypes in an unsafe manner in their program. They mix signed and unsigned numeric types in arithmetic operations. This can allow an attacker to launch an integer overflow attack and eventually overflow a buffer. A developer can apply Safe Library Replacement (section 7.4.3) transformation to replace unsafe functions. The mixing of unsafe and safe datatypes can be corrected by applying Explicit Type Enforcement (section 7.4.2) transformation.

3. Developers use direct pointer arithmetic when they are handling buffers. The direct pointer manipulation can lead to a buffer overflow vulnerability. A developer can apply Safe Type Replacement (section 7.4.4) transformation to replace array-based buffer representation with safe datatypes that buffer bound information.

A Decorated Filter transformation (section 7.4.1) can be applied in all three cases to rectify an input variable. A combination of Partitioning (section 6.1.1) and Least Privilege (section 6.1.2) transformations can be applied to partition and program and run them with minimum privilege level, so that attackers are not motivated to exploit a buffer overflow vulnerability. A program may have an error that leads to buffer overflow attacks, but since it is running at lower privilege level, an attacker is not able to gain much even after exploiting the vulnerability.

The next sections describe these methods in detail. We will illustrate the transformations with simple programs. We are assuming that the developer is using the Eclipse C development environment and that program transformation tools are available.

12.1.1 Replacing Unsafe Libraries

A Safe Library Replacement transformation (section 7.4.3) can replace the following unsafe library functions: strcpy, strcat, memcpy, memmove, gets, getenv, printf, scanf etc. We will describe these functions next and show how the library replacement program transformation secures a program. A list of these functions and their safe alternatives is available in section 7.4.3.
A developer has to select the *Safe Library Replacement* transformation tool in Eclipse and select the unsafe functions to replace as well as the safe alternatives. The tool will attempt to replace the instances of unsafe functions.

**strcpy, strcat and gets.** `strcpy` and `strcat` functions take two parameters: the source buffer and the destination buffer. Suppose the developer chooses `g_strlcpy` and `g_strlcat` from glib library as safe alternative functions. These functions take a third parameter: the length of the destination buffer. The program transformation tool has to determine the length of the destination buffer when it introduces the new function. Two C library functions, `sizeof` and `malloc_usable_size`, can be used to calculate the size of stack buffers and heap buffers correspondingly.

Consider the following C program that has buffer overflow and integer overflow errors.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char * argv[])
{
    char dest1[12] = "Hello ";
    char *dest2 = (char *) malloc (sizeof(char) * 12);
    char *src = (char *) malloc (sizeof(char) * 12);
    char *memdest = (char *) malloc (sizeof(char) * 20);
    char *memsrc = (char *) malloc (sizeof(char) * 10);
    signed int count;

    printf("Enter your name:");
    gets (src);
    printf ("\nEnter your name again:");
    scanf ("How many bytes to copy?:");
    if (count <= malloc_usable_size (memsrc))
        memcpy (memdest, memsrc, count);
    printf("\n", memdest);
    return 0;
}
```

The program has unsafe `strcpy` and `strcat` functions in lines 15, 16, and unsafe `gets` function used in line 14 and 20. The tool will replace these functions. In lines 15 and 16, it will introduce `g_strlcat` and `g_strlcpy`. These functions take the length of destination buffer as an additional parameter. By taking a backward trace, the tool can determine that `dest1` in line 15 is stack allocated (line 6), and `dest2` in line 16 is heap allocated (line 7). The tool uses `sizeof` function for stack allocated and `malloc_usable_size` function for heap allocated buffers.
Similarly, the tool will replace the \texttt{gets} function with \texttt{fgets} function. \texttt{fgets} takes two additional parameters: the maximum number of characters to be read (i.e. the size of the buffer), and a pointer to a \texttt{FILE} object that identifies the stream where characters are read from. The tool uses \texttt{sizeof} and \texttt{malloc\_usable\_size} functions for size calculation, and \texttt{stdin} as the \texttt{FILE} pointer. Here is the transformed program segment,

\begin{verbatim}
... 
13  fgets (src, malloc_usable_size (src), stdin); 
14  g_strlcat (dest1, src, sizeof (dest1)); 
15  g_strlcpy (dest2, src, malloc_usable_size (dest2));
16  printf ("%s %s", dest1, dest2); 
17  printf("\nEnter your name again: ");
18  fgets (memsrc, malloc_usable_size (memsrc), stdin); 
... 
\end{verbatim}

\texttt{memcpy} and \texttt{memmove}. \texttt{memcpy} and \texttt{memmove} takes 3 parameters: the source buffer, the destination buffer and the number of bytes to be copied (typically the size of the source buffer). The tool will introduce a safe version of the function in line 24 of the original program.

\begin{verbatim}
... 
#define min(a,b) (a<b)?a:b
#define memcpy_s(a,b,c,d) memcpy(a,c,min(b,d))
... 
if (count <= malloc_usable_size (memsrc))
  memcpy_s (memdest, malloc_usable_size (memdest), memsrc, count);
... 
\end{verbatim}

In this case, the macro takes the minimum of the two buffer sizes, and uses it to copy buffer. As a result, even if \texttt{memdest} is smaller than \texttt{memsrc}, it will not be overflown.

\texttt{getenv}. \texttt{getenv} function returns a pointer to a string associated with the matched list member. The string is the value of an environment variable. There is a possibility of buffer overflow, if a program calls \texttt{getenv} multiple times. The string pointing to the result of a \texttt{getenv} call may be overwritten by a subsequent call to the \texttt{getenv} function. The pointer can also be invalidated as a result of changes made to the environment list through calls to \texttt{putenv()}, \texttt{setenv()}, or other means. The tool uses the \texttt{getenv} call but surround it with guard code that does not allow buffer overflow. For each \texttt{getenv} call, the tool will copy the result to another buffer so that it does not get overwritten.
Consider the following C program [taken from CERT repository],

```c
... char * tmpvar; char * tempvar; tmpvar = getenv("TMP"); tempvar = getenv("TEMP"); ...
```

This code example is noncompliant because the string referenced by `tmpvar` may be overwritten as a result of the second call to the `getenv()` function. As a result, it is possible that both `tmpvar` and `tempvar` will compare equal even if the two environment variables have different values. If the size of `tmpvar` is smaller than the possible size of `tempvar`, then there is a possibility of buffer overflow.

```c
... char * tmpvar; char * tempvar; tmpvar = getenv("TMP"); tempvar = getenv("TEMP");
const char * temp = getenv("TMP");
if (temp != NULL) {
  tmpvar = (char *)malloc(strlen(temp)+1);
  if (tmpvar != NULL)
    g_strlcpy(tmpvar, temp, malloc_usable_size (tmpvar));
  else
    /* Handle error */
} else
return -1;

temp = getenv("TEMP");
if (temp != NULL) {
  tempvar = (char *)malloc(strlen(temp)+1);
  if (tempvar != NULL)
    g_strlcpy(tempvar, temp, malloc_usable_size (tempvar));
  else {
    free(tmpvar);
    tmpvar = NULL;
    /* Handle error */
  }
} else {
  free(tmpvar);
  tmpvar = NULL;
  return -1;
}
```

printf and scanf. There are many variants of printf and scanf functions, all vulnerable to buffer overflow attacks. Suppose the developer selects fgets as the safe alternative function.
Consider the following program [Example from the CERT advisory],

```c
long sl;
if (scanf("%ld", &sl) != 1) {
    /* handle error */
}
```

The tool replaces the `scanf` function with `fgets` to input a string and `strtol` to convert the string to an integer. Error checking is provided to make sure that the value is a valid integer in the range of `long`.

```c
char buff[25];
char *end_ptr;
long sl;
if (fgets(buff, sizeof(buff), stdin) == NULL) {
    if (puts("EOF or read error \n") == EOF) {
        /* Handle error */
    }
} else {
    errno = 0;
    sl = strtol(buff, &end_ptr, 10);
    if (ERANGE == errno) {
        if (puts("number out of range") == EOF) {
            /* Handle error */
        }
    }
    ...
```

### 12.1.2 Enforcing Correct Integer Operation

Lines 11, 23 and 24 of the C program listed at the start of section 12.1.1 contribute to an integer overflow vulnerability. Line 11 declares `count` as a signed integer variable. It is used in line 23 in a comparison operation. However, it is compared to a `size_t` variable which is the return value of the `malloc_usable_size` function. `size_t` corresponds to an unsigned value. An attacker can bypass the comparison by passing a negative value. Then `count` is used to denote the buffer size in a memory copy in line 24. The `memcpy` function accepts a `size_t` parameter in that position; hence, a signed value is used in an unsigned context. The signed value will be promoted to a corresponding large unsigned value and the `memdest` buffer will be overflown.

Replacing the `gets` function in line 14 does limit an attacker’s ability to overflow the `memdest` buffer. However, defense in depth principle suggests that the integer overflow opportunity should be removed, especially to protect the any previous content of the `memdest` buffer. A tool will introduce explicit type casts to prevent the integer overflow vulnerability.
if ((unsigned) count <= malloc_usable_size (memsrc))
    memcpy (memdest, memsrc, (unsigned) count);

12.1.3 Replacing Pointers with Safe Data Structure

Not all buffer overflows originate from the use of unsafe functions. It is a common practice to directly copy bytes from one buffer to another. A Safe Type Replacement (section 7.4.4) transformation can be applied to replace the unsafe representation of buffers with a safe representation. A tool can replace char * pointers with a safe data buffer such as the stralloc data structure. stralloc [93] is of struct with three fields: s is a char * pointer that stores the buffer, a is an integer that represents the memory allocated, and len is an integer that represents the size of the buffer that has been used. Thus, stralloc is just a wrapper for a char * pointer with length information. The length information can be used to only allow operations that do not overflow a buffer.

Consider the following C program that has a buffer overflow error in line 8 of the foo function, since the size of the src buffer is larger than the size of the destination buffer.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void foo(char * dst, char *src) {
    int i = malloc_usable_size(src);
    while (i--) {
        *dst = *src;
        ++dst;
        ++src;
    }
    *dst = '\0';
}

int main (int argc, char *argv[]) {
    char *src = (char *) malloc (sizeof(char) * 100);
    char *dst = (char *) malloc (sizeof(char) * 20);
    _strlcpy (src, argv[1], malloc_usable_size (src));
    foo (dst, src);
    printf("%s %s\n", src, dst);
    return 1;
}
```

A tool can search for direct pointer operations in a program, and check whether it is vulnerable by tracing backward the source and target of the operation. It will find that the source buffer is coming from an external user (line 18), which is unsafe. Hence it will replace the representation of the target buffer dst.
In the modified program, the `dst` variable has been replaced with `newvar1`, a `stralloc` data type (line 6-8). Then the copy to `dst` is replaced with `newvar1.s` (line 12). The length information of the `newvar1` is maintained and checked at each step (line 11-16, 20-25). Finally, the buffer is copied back to the `dst` variable (line 27). The code snippet follows a pattern that is used to replace direct pointer operations.

```c
... include "stralloc.h" void foo(char* dst, char* src) {
  int i = malloc_usable_size(src);
  stralloc newvar1 = {0};
  if (!stralloc_ready(&newvar1, malloc_usable_size(dst))) exit(0);
  stralloc_copys (&newvar1, dst);
  signed int newcounterintvar1 = 0;
  while (i--) {
    if (newcounterintvar1 < (signed)(((signed) newvar1.a) - 1)) {
      newvar1.s[newcounterintvar1] = *src;
      if ((unsigned) newcounterintvar1 >= newvar1.len)
        newvar1.len = (unsigned) newcounterintvar1 + 1;
    } else
      break;
    newcounterintvar1++;
++src;
  }
  if (newcounterintvar1 < (signed)(((signed) newvar1.a) - 1)) {
    newvar1.s[newcounterintvar1] = '\0';
    if ((unsigned) newcounterintvar1 >= newvar1.len)
      newvar1.len = (unsigned) newcounterintvar1 + 1;
    newcounterintvar1++;
  }
  stralloc_0 (&newvar1);
  g_strlcpy (dst, newvar1.s, malloc_usable_size(dst));
}
...  
12.1.4 Input Rectification and Other Techniques

Another approach to protect against buffer overflow attacks is to apply input rectification filters. A tool can automatically detect the places that require input rectification by dataflow analysis and taint analysis from sink to source. For example, it is possible to detect potential integer overflow errors by performing a sink-to-source analysis. The possible sinks are various functions that use length parameter, such as `memcpy`, `bcopy`, `malloc` etc. If the variables used in these sinks are traced back to a source that comes from external input, the integer variable has to be rectified.

The rectification filters can be a function that applies replication policies. In fact, adding a type cast by
Explicit Type Enforcement (section 7.4.2) transformation is also a form of input rectification. Consider the C program that we have used to illustrate buffer overflow vulnerability in section 12.1.1. It has an integer overflow vulnerability in lines 11, 23-24. A tool can determine the integer context in line 23, and appropriately type cast following the MISRA C guideline [142]. Another way is to apply a filter that uses taint information (or dataflow analysis) and forces an integer to be used appropriately. Both the modifications are shown here,

```
... (unsigned) count <= malloc_usable_size (memsrc))
memcpy (memdest, memsrc, force_unsigned (count));
...```

The tool will be able to determine that `count` is a signed integer; it will apply a checker function `force_unsigned` as a filter that forces the value of `count` to be within the appropriate range.

Most of the program transformations to prevent buffer overflow vulnerability make small, frequent changes to the code. It is also possible to apply two fundamental security-oriented program transformations, Partitioning (section 6.1.1) and Least Privilege (section 6.1.2), to passively prevent buffer overflow attacks.

### 12.2 Preventing SQL Injection Vulnerability in PHP Programs

MySQL is currently the most popular open source database server in existence. It is very commonly used in conjunction with PHP scripts to create powerful and dynamic server-side applications. Because of the popularity and wide-spread use, PHP-based web applications are the most common target. In 2000, PHP-related vulnerabilities were 1.9% of the total vulnerabilities listed in the national vulnerability database [49]. In 2009, they contribute to 29.9% of all vulnerabilities reported. These allow attackers to steal or destroy data from data sources linked to the webserver (such as an SQL database), send spam or contribute to DoS attacks using malware, which itself can be installed on the vulnerable servers.

Most of the PHP vulnerabilities are SQL injection and cross-site scripting vulnerabilities. In this section, we will focus on two most common programming mistakes that contribute to SQL injection vulnerabilities.

1. PHP language has new libraries that use pre-compiled SQL statements. These libraries prevent SQL injection attacks by identifying tainted input in the data part of the SQL statement. A developer can
apply Safe Library Replacement (section 7.4.3), and Safe Type Replacement (section 7.4.4) transformations to introduce the newer library functions.

2. Developers do not sufficiently check input data before using them in sensitive functions. From the list of 7601 PHP vulnerabilities from 2000 to 2008, only 964 are properly tagged, i.e. one can found detailed information about the type of vulnerability. Among these, 790 (81.9%) are directly or indirectly related to a lack input of validation (code/command/SQL injections, cross site scripting, path traversal) [49]. A developer can apply Decorated Filter (section 7.4.1), Error Message Suppression (section 10.5.1) and Exception Shielding (section 10.5.2) transformations to add proper input validation.

The next sections describe these methods in detail. We will illustrate the transformations with a simple PHP program. We are assuming that the developer is using the Eclipse environment and that program transformation tools are available.

### 12.2.1 Replacing Unsafe Libraries

The typical way for PHP applications to communicate with MySQL databases is to use the MySQL API. It has been extended with the MySQLi (MySQL improved) API that uses pre-compiled SQL statements. A combination of Safe Library Replacement (section 7.4.3), and Safe Type Replacement (section 7.4.4) transformations can help developers replace the old API.

We list a simple example that shows how to connect, execute a query, print resulting rows and disconnect from a MySQL database. The example is an excerpt from a PHP application that is the front end of a news database. We will list two PHP files. The first is global.php that connects with a MySQL database named news. Lines 3-6 of the program describes the connection variables, line 8 creates a connection, and line 9 selects the database.

```php
<?php
 //set database connection variables
 $hostname='127.0.0.1';
 $username='root';
 $password='passwd';
 $database='news';

 $connect = mysql_connect($hostname,$username,$password); //establish database connection
 mysql_select_db($database); //select database
 ?>
```
The second file is index.php. It typically lists all the news entries in the database (the result of the SQL query in line 9). If the author parameter is specified (line 4), then it uses the parameter to query the database (5-7). In both cases the results are listed in lines 17 through 26.

```php
<?php
include 'global.php';

if (isset($_GET['author'])) {
    $author_name = $_GET['author'];
    $query_string=stripslashes("SELECT id,title FROM news WHERE author='$author_name'"));
    $result = mysql_query($query_string);
} else {
    $result = mysql_query("SELECT id,title FROM news");
}
?>
<html>
...
...
...
<table width="90%" cellpadding="3" cellspacing="0" border="0">
<?php
while($row=mysql_fetch_array($result)) {
    print '<tr>
<td>'.$row['title'].'</td>
<td><a href="articles.php?id='.$row['id'].'">edit</a > | <a href="'.$_SERVER['PHP_SELF'].'?id='.$row['id'].'">delete</a></td>
</tr>
';
}
</table>
</html>
```

An attacker can bypass the query in line 5, and force the program to list all the news entries. Any non-trivial application will only want to list only the news entries that a user is allowed to see. Therefore, it will include authentication, and access control. We strip that part to keep the program simple.

Suppose a developer wants to apply Safe Library Replacement and Safe Type Replacement transformations. He can select the MySQL connection variable in line 8 of global.php and apply a Safe Type Replacement tool to replace it with MySQLi objects. The program transformation will create a replica of global.php which uses MySQLi objects.

```php
...
$connect = new mysqli($hostname,$username,$password); //establish database connection

...```

Then the developer applies the Safe Library Replacement transformation on global.php to replace the mysql_select_db function in line 9. He or she will have to specify the new connection object.

211
Then, the developer can apply multiple Safe Type Replacement and Safe Library Replacement transformations on index.php. The program transformation introduces a MySQLi_STMT object in line 6 and 11, and replaces MySQL functions with MySQLi functions to use the new datatype.

The program transformation is an example of a composite refactoring, in which the program behavior will not be the same in some intermediate steps. The good path behavior of the program remains the same at the end of the program transformation.

12.2.2 Applying Filters for Input Rectification

Changing the datatype and library is a complex transformation that might not be possible in all circumstances. Another approach is to apply the Decorated Filter transformation to add input validation filters.

We will illustrate the program transformation on the index.php program listed in section 12.2.1. Suppose a developer selects the author_name variable in line 5 of the program as the target of a Decorated Filter
transformation. A tool presents a list of SQL-input validation filters to the developer. Suppose the developer selects a filter that strips SQL OR statement from user input. The program transformation tool will add the filter objects to the existing codebase (if they are not already present), and use them in the program as part of a Decorator pattern [74]. The modified code is listed here.

```
<?php
include 'global.php';
include 'abstract_container.php';
include 'exceptions.php';

if (isset($_GET['author'])) {
    $author_name = new unsafe_string($_GET['author']);
    $policy_decorator = new sql_policy_strip_or($author_name);
    $query_string=stripslashes("SELECT id,title FROM news WHERE author=
    '{\$policy_decorator->convert()->get_str()}');
    $result = mysql_query($query_string);
    ...
}
```

The developer can then select the code block that has been modified (lines 9-12) and apply the Exception Shielding transformation.

```
try{
    $query_string=stripslashes("SELECT id,title FROM news WHERE author=
    '{\$policy_decorator->convert()->get_str()}');
    $result = mysql_query($query_string);
} catch (MaliciousModifiedException $e) {
    print ("Message: {\$e->getMessage()}";
} catch (Exception $e){
    print ("Message: {\$e->getMessage()}";
}
```

In order to prevent blind SQL injection, the developer can then apply the Error Message Suppression transformation to generalize the error messages, so that they do not reveal internal information.

12.3 Conclusion

In this chapter, we have described two most common vulnerabilities faced by developers and how program transformations can be composed to prevent them. The goal of a secure system developer is to have defense in depth. Composing program transformations helps a developer attain that goal.
Chapter 13

Usefulness of the Program Transformation Catalog

How useful is our catalog of program transformations? Two factors can indicate its usefulness. First, our catalog should contain program transformations that cover the problem domain, i.e. solve various types of security problems. Second, the program transformations in our catalog should be used often by developers. In this chapter, we will argue how our catalog of program transformations achieves the first goal. Achieving the second goal involves tooling the program transformations in the catalog and extensive user study with the tools. We will also describe our efforts in this regard.

13.1 Coverage of Program Transformations

Making a quantitative analysis of what percentage of security problems does our catalog cover is hard because the definition of security and security solutions is fuzzy. There might be multiple factors that cause a security problem. For example, a buffer overflow vulnerability points to improper input validation and the use of unsafe buffer-handling functions. But a successful exploit of a buffer overflow also requires elevation of privilege vulnerability to be present. Hence, a buffer overflow vulnerability is a sign of multiple weaknesses present in a system. Consequently, the solution to mitigate a security vulnerability should plug in various problems. Security is an emergent property; it is unlikely that one can add security to a system by adding a component or by modifying code in one part of a system. Hence, any one particular program transformation does not solve a security problems; it only contributes to the overall security of the system. Chapter 12 describes how multiple program transformations can be used together to remove buffer overflow vulnerability.

Nevertheless, transformations in our catalog cover the most relevant security problems faced by today’s developers. Our program transformation catalog originates from the analysis of both the problem domain and the solution domain of security. We have analyzed various reports on vulnerability trend analy-
sis [45] [151] [196] [206] [207] to identify the most relevant security problems. On the other hand, our comprehensive security pattern catalog [92] compiles the solutions that experts use to solve security problems. The program transformations mostly come from the security pattern catalog as well as from well-known security solutions that can be automated. Combining these two dimensions ensure that our catalog includes transformations that provide real solutions for real problems.

The vulnerability trend reports indicate that the security problems faced by developers can be classified into 5 categories: improper input validation vulnerabilities, authentication vulnerabilities, access control vulnerabilities, error handling vulnerabilities and denial of service vulnerabilities. We have surveyed the Bugtraq [34] vulnerability list on several random weeks over a period of more than a year and found the same trend. Table 13.1 lists the vulnerabilities reported in the Bugtraq database on 5 random weeks, and their distribution.

In the first week of September 2008, 66% of the vulnerabilities in that week were some form of input validation problems, 2% were related with bad authentication and bad access control, 11% were denial of service problems, 7% were error handling problems, and 14% were miscellaneous other problems or combinations of the the aforementioned problems. In reality, the partitioning is not mutually exclusive. For example, 14 out of 22 vulnerabilities in the other category are temporary file creation vulnerabilities. These can be removed by *chroot Jail* (section 7.5.1) or *Safe library Replacement* (section 7.4.3) transformations. If we put them under input validation category then Other category has 8 vulnerabilities (5%). Also input validation then has 116 (75%) vulnerabilities.

A security vulnerability is reported according to the manifestation of the vulnerability; the root cause

<table>
<thead>
<tr>
<th></th>
<th>Sep 2008</th>
<th>Jan 2009</th>
<th>May 2009</th>
<th>Sep 2009</th>
<th>Jan 2010</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Validation</strong></td>
<td>102 (66%)</td>
<td>63 (61%)</td>
<td>37 (58%)</td>
<td>53 (44%)</td>
<td>89 (79%)</td>
</tr>
<tr>
<td><strong>XSS</strong></td>
<td>21 (21%)</td>
<td>13 (21%)</td>
<td>9 (24%)</td>
<td>8 (15%)</td>
<td>25 (28%)</td>
</tr>
<tr>
<td><strong>Buffer Overflow</strong></td>
<td>28 (28%)</td>
<td>5 (8%)</td>
<td>11 (29%)</td>
<td>25 (47%)</td>
<td>13 (15%)</td>
</tr>
<tr>
<td><strong>SQL Injection</strong></td>
<td>34 (33%)</td>
<td>41 (65%)</td>
<td>7 (19%)</td>
<td>6 (11%)</td>
<td>26 (29%)</td>
</tr>
<tr>
<td><strong>Authentication</strong></td>
<td>0 (0%)</td>
<td>5 (5%)</td>
<td>3 (5%)</td>
<td>1 (1%)</td>
<td>2 (2%)</td>
</tr>
<tr>
<td><strong>Authorization</strong></td>
<td>3 (2%)</td>
<td>3 (3%)</td>
<td>5 (8%)</td>
<td>10 (8%)</td>
<td>7 (6%)</td>
</tr>
<tr>
<td><strong>Error Handling</strong></td>
<td>10 (7%)</td>
<td>3 (3%)</td>
<td>4 (6%)</td>
<td>12 (10%)</td>
<td>2 (2%)</td>
</tr>
<tr>
<td><strong>Denial of Service</strong></td>
<td>17 (11%)</td>
<td>7 (7%)</td>
<td>9 (14%)</td>
<td>27 (23%)</td>
<td>7 (6%)</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td>22 (14%)</td>
<td>22 (21%)</td>
<td>6 (9%)</td>
<td>17 (14%)</td>
<td>6 (5%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>154</td>
<td>103</td>
<td>64</td>
<td>120</td>
<td>113</td>
</tr>
</tbody>
</table>

Table 13.1: Vulnerability Trends in 5 Random Weeks
might be different from the final manifestation. For example, 33% of the denial of service problems in the survey can be solved by some sort of input validation. Our program transformation catalog contains transformations that can be used to fix all these different types of vulnerabilities (not the actual instances of vulnerabilities). There were 28 instances of buffer overflow vulnerabilities affecting software from 22 different vendors. We could not analyze 9 instances that affect proprietary software. 17 of the remaining 19 instances can be solved by applying the Safe Library Replacement transformation (section 7.4.3). In the remaining two cases, buffer overflow vulnerabilities originate from direct manipulation of pointers. The Safe Type Replacement transformation (section 7.4.4) from our catalog could remove these vulnerabilities. Also, there were 34 SQL injection and 21 cross-site scripting attack instances that could be fixed by Decorated Filter transformation.

Other problems originating from unvalidated inputs (injection attacks) and improper error handling can also be solved by transformations in our catalog. The authentication and authorization related transformations in the catalog can be used to add new components, or strengthen existing components, that can eradicate most of the problems originating from broken authentication and broken access control.

Another aspect is that not all transformations in our catalog has a one-to-one relationship with vulnerabilities. Many security problems have their roots in bad authentication and access control, but they manifest in other ways. Transformations to add an authentication, authorization component, or a partitioning transformation does not solve any one problem; they prevent other vulnerabilities from occurring. Hence the protection provided by our catalog is stronger than that suggested by the coverage figures.

The program transformations in our catalog come from various sources. Their main source is the pattern catalog, which in turn is the collective knowledge of many security experts. Hence the solutions introduced by the program transformations cover various types of security problems.

### 13.2 Adoption of Program Transformations

In order to evaluate how often a security solution is used, or which transformations are more useful than others, one has to build tools for all of them. Then again, the tools will be platform specific and programming language specific, factors that might impact their usability. We believe that it is unreasonable to expect researchers to build tools for every platform. Building a tool for a single application is usually not cost effective, so it is unreasonable to expect application developers to do it either. It will need to be done by
platform and tool vendors, who can amortize the cost over many users. Only that will provide a useful framework on which a usefulness study can be done.

It is easy to build tools for some program transformations; for a few others, only parts of the transformation can be automated. It is natural that the first tools will be the ones that are the easiest to build. However, the importance of these tools cannot be minimized, because these tools target the most important security vulnerabilities. Table 13.1 indicates that input validation vulnerabilities are the most prominent, especially buffer overflow vulnerability, SQL injection vulnerability and cross site scripting vulnerability. Chapter 12 describes how program transformations can be applied to remove the first two vulnerabilities\(^1\). These program transformations make small changes at each program point, but the changes occur at many points. Tools that make small, structural changes are easy to build, and the frequency of the changes suggests that it can be tedious for a human to make them consistently without tool support.

\[\begin{array}{ll}
\text{Add Audit Interceptor} & \text{Message Digest Creator} \\
\text{Add Replicated System} & \text{Message Intercepting Gateway} \\
\text{Checkpointed System} & \text{Partitioning} \\
\text{chroot Jail} & \text{Randomization} \\
\text{Controlled Process Creation} & \text{Resource Management} \\
\text{Decorated Filter} & \text{Safe Library Replacement} \\
\text{Encryption/Decryption} & \text{Safe Type Replacement} \\
\text{Error Message Suppressor} & \text{Secure Logger} \\
\text{Exception Shielding} & \text{Signature Generation} \\
\text{Explicit Type Casting} & \text{Single Access Point} \\
\text{Fuzzing} & \text{Unique Location for Each WRITE Request} \\
\text{Guarded Object} & \\
\text{Add Perimeter Filter} & \text{Policy Enforcement Point} \\
\text{Add Authentication Enforcer} & \text{Add Authorization Enforcer} \\
\text{Add Password Synchronizer} & \text{Add Tandem System} \\
\text{Add Standby} & \text{Credential Tokenizer} \\
\text{Least Privilege} & \text{Message Caching} \\
\text{Secure Resource Pooling} & \text{Secure Session Object} \\
\text{Secure Sign On Delegate} & \\
\end{array}\]

![Feasibility of Building Program Transformation Tools](image)

**Figure 13.1: Feasibility of Building Program Transformation Tools**

Figure 13.1 lists our program transformations based on two criteria: how feasible is it to build tools

\(^1\)Cross site scripting vulnerability can be prevented by applying *Decorated Filter*(section 7.4.1) transformation

217
for program transformations, and how much effort is needed to apply the solution manually? The top-left quadrant lists the largest number of program transformations, including the program transformations that prevent injection attacks. Most of these program transformations make small, but numerous changes. However, some program transformations such as Single Access Point and Partitioning are also listed. We described how most parts of these transformations can be automated by reusing simple refactoring tools and other remoting tools. A Partitioning transformation first distributes program components and may create new components; developers can apply refactoring tools that extract classes, split classes and move methods to introduce these changes step by step. Developers have to determine a partitioning strategy beforehand; this step can be aided by architecture visualization tools. Once the partitioning is done, some local method calls are replaced by remote method calls. There are existing tools that automate these steps to improve performance [201, 203].

We have been developing proof-of-concept implementations for some program transformations in this category. Our current and future effort involves developing more user-friendly tools for various program transformations in our catalog. We have been developing these tools for C, Java and PHP languages in the Eclipse IDE. We will apply the tools to remove various types of input validation vulnerabilities reported in the Bugtraq database; we also plan to perform extensive user studies.

The top-right quadrant lists program transformations that are difficult to perform manually, and are difficult to automate. These program transformations introduce a lot of change. These changes require a lot of behavioral information. These program transformations are more interactive than the ones in the top-left quadrant; developers perform some steps manually and use automated tools to perform some of the steps. Consider the Add Authentication Enforcer transformation that composes an authentication component using an existing framework. The default authentication component must be customized by developers to fit their application; this involves specifying authentication type (username/password), source of user inputs, authentication knowledge base (username/password store), and the outcome of a successful authentication (which principals and credentials are added to the subject), etc. Many of these steps can be automated, but others remain manual.

The bottom-left quadrant lists two program transformations that perform small changes in a few places. These program transformations are easy to automate, but the tools may not be useful enough, since the same change can be easy to introduce manually.
The bottom-right quadrant contains one program transformation that is easy to perform manually, but is hard to automate. Add Account Lockout transformation restricts authentication attempts to a few tries to prevent a brute-force attack against authentication tokens. It might be as simple as adding a for loop with conditional clause. But finding where to insert the code is difficult if the authentication component is implemented arbitrarily. A tool for this transformation is unnecessary because of the small change the transformation makes in very few points in a program.

Our program transformations involve multiple steps; many of these steps are connector transformations. Spitznagel et.al. [193] describes a small set of systematic connector transformations. Figure 13.2 describes how program transformations in our catalog correspond to connector transformations. We can reuse the connector transformations in their work to implement steps of our program transformations.

Figure 13.2: Security-oriented Program Transformations That Involve Connector Transformations
Spitznagel et.al. [193] describes transformations on connector types such as RPC or data streams. Some of our security-oriented program transformations add new component or distribute components; these modify the connectors between components. An example is the Add Authentication Enforcer transformation that involves an Add Redirect connector transformation. The transformation redirects data flow through an authentication component. The redirection can be to a remote authentication component or a local method call (or message passing) to a local authentication component. However, the authentication component has to be defined first. An Add Authentication Enforcer transformation also involves steps to compose a default authentication from an authentication framework.

Similarly, a Safe Library Replacement transformation can be viewed as an Add Redirect connector transformation. Calls to an old library are redirected to a new library. The two libraries typically have different API which means that the transformation has to be aided by various control-flow analysis and data-flow analysis techniques.

Some program transformations combine multiple connector transformations. A Replicated Component transformation creates a replica of a component and a workload management proxy component; a combination of Add Switch and Aggregate transformations realize this.

There are program transformations in our catalog that do not involve any connector transformations. An example is a chroot Jail transformation that constrains a process inside a chroot jail. The program transformation modifies the environment of a process and creates a jail environment for it.

Two studies of the way refactoring tools are used [146] [145] report that the most popular refactorings are Rename, Extract Local Variable, Inline, Extract Method, and Move. Of these, Rename, a very simple refactoring, is by far the most popular. Note that all the cited refactorings are fairly small. If these results can predict what are likely to be the most popular security-oriented program transformations, they will probably be the simplest transformations, from the Small impact category.

Tools will be beneficial in the long run; but the real value of the catalog is the documentation of the systematic process to perform a program transformation. In many cases, some steps will be automated, while others will remain manual: many of the larger program transformation that introduce authentication components are examples of such transformations. Nevertheless, program transformations are valuable in their automated and semi-automated form.
13.3 Conclusion

Since new security attacks are being discovered and launched all the time, the security solutions to prevent the attacks will need to change, and so will need the corresponding security-oriented program transformations. Our current catalog covers most of the important security problems faced by today’s developers. We plan to continue maintaining and extending our catalog in future.
Chapter 14

Conclusion

Security-oriented program transformations show how to think about security systematically. Since new kinds of security threats continue to appear, even systems that are currently being built to high security standards will eventually need to be changed, and so will need security-oriented program transformations. It is usually easier to solve a particular problem than to come up with a general solution to all problems of a particular class. However, there are many fewer general security solutions than there are security flaws. Applying security solutions defined as transformations will be cheaper in the long run than fixing every security flaw individually.

In this section we will describe some future directions of our work and project our vision.

14.1 Open Research Questions

Treating security solutions as program transformations leads to a lot of research questions. The most obvious is how to implement each transformation. A deeper question is what share of security solutions can be implemented as transformations. Are there specific properties of a security solution that enable automation?

Security-oriented program transformations should be composable. The order of composition is strict for some transformations, while other transformations can be composed in any order. For example, a Partitioning transformation must be applied before a chroot Jail transformation, but one can apply a Safe Library Replacement transformation in any order. A pattern language that organizes security solutions [92] can provide a guideline for the order of composing corresponding program transformations.

Another open question is finding a suitable platform for implementing program transformations. Possibly the easiest way to implement transformations that weave in method calls is to use aspect-orientation [117]. Logging and weaving in method calls to new components are the poster children of AOP. So transformations that add a module, such as the Add * transformations, can be implemented with AOP relatively easily.
On the other hand, program transformations that rearrange the structure, such as Partitioning, are more like refactorings, and are hard to implement using aspects. All these transformations can probably be implemented by a code transformation language such as TXL [51] and Stratego/XT [211], or a framework for implementing refactorings. A more philosophical question is whether a domain-specific language benefits the development of transformation tools.

Ideally, applying a program transformation should have similar effect as applying a refactoring: it makes a program more structured and more comprehensible. Realistically, the code automatically generated by security-oriented program transformations can make a program hard to understand and maintain. The success of security-oriented program transformations may depend on how they beautify automatically generated code. However, the increase in productivity of users of automated transformations can offset their discomfort with automatically generated code. Finding the sweet spot remains an open research problem.

14.2 Future Work

There are scopes of building on my current work as well as expanding my work into newer terrains. Here are some research areas that I will like to concentrate.

Catalog of Program Transformations. Our catalog will grow – in response to new classes of vulnerabilities – and pro-actively, as more people use it and begin to look for missing pieces. The next important step in exploring security-oriented program transformations will be to study how different transformations are related. A developer should apply multiple program transformations at many levels of a program to have ‘defense in depth’. Exploring the relationship between program transformations help use understand how multiple program transformations can be composed—whether they can be composed haphazardly or whether they require a certain order.

At the same time, it is possible to study specific programming platforms and identify suitable program transformations. A research project of particular interest will be to concentrate on security vulnerabilities introduced by various parallel programming paradigms, and concentrate on corresponding program transformations.
Program Transformation Tools for Eclipse IDE. We have built a few prototypes of transformation tools and applied them to remove real vulnerabilities from real programs. Currently, we are working on four tools with several undergraduate and graduate students at UIUC; these program transformations are applicable to C, Java and PHP programs on Eclipse IDE. By building tools and performing user studies with the tools, we will be able to answer the question of how useful and applicable the security-oriented program transformations are.

For many kinds of security threats, we already know security-oriented program transformations that can automatically fix programs to eliminate these threats. There are certainly more security threats to study. However, the program transformations are always platform specific, so it is unreasonable to expect researchers to build them for every platform. Building a tool for a single application is usually not cost effective, so it is unreasonable to expect application developers to do it. It will need to be done by platform and tool vendors, who can amortize the cost over many users.

Program Analysis to detect ‘Security Smells’. Program transformation tools make structural changes without deep understanding of program behavior. The behavioral knowledge comes from manual specifications. The most important behavioral specification is specifying the program points where a program transformation is applied. A developer has to detect the ‘security smells’ in a program and apply program transformations at those points.

The most common ‘security smell’ is a security vulnerability that needs to be fixed. There has been a rich vein of research on vulnerability detection tools and technologies. Our work on program transformations benefits from this research because an automated detection tool can act as a guide for a developer and suggest him/her where to apply a program transformation. We want to marry the research on detecting a vulnerability with the research on fixing it. For security vulnerabilities that do not have detection tools, we want to explore attack patterns so that smart detection tools can be built.

Not all ‘security smells’ are related with vulnerabilities. Hence, some program transformations do not directly fix vulnerabilities. For example, a Single Access Point program transformation minimizes the access points of a system. In this case, the ‘security smell’ is the existence of many access points at many parts of a program. These smells are part of common folklore, but there have been no effort to define them and detect them in programs. Research on program comprehension and program analysis techniques are helpful in building detection tools to guide secure system developers.
Another potential research project is an empirical study of the frequency of ‘security smells’. I will be interested to do this study to determine which program transformation tools will be used more frequently than others.

14.3 Onward!

Keeping systems secure requires constantly improving them. Security-oriented program transformations are a repeatable, systematic approach to eradicating system vulnerabilities during program development and during maintenance. Security-oriented program transformations have the potential to enable security engineers to develop policies and let a tool apply those policies. This will blend human creativity with the thoroughness of the computer.
Appendix A

Catalog of Security-oriented Program Transformations

Each transformation is summarized in three sentences.

1. The first sentence describes the problem resolved by the transformation.

2. The second sentence describes what a developer needs to specify (the policy).

3. The last sentence summarizes the structural changes that are made by transformation (the mechanism).

The name of a program transformation includes a reference to the source that first described this security solution. There are some program transformations that do not originate from security patterns; some of these are program transformations that introduce a well-known security feature (e.g. encryption, message digest etc.) while others are new program transformations. These do not have any references.

The catalog currently has 37 program transformations. They are listed in alphabetical order.

**T1. Add Account Lockout** [118]. How can you prevent an attacker from guessing passwords?

A developer specifies the authentication component where the lockout policy is enforced. The transformation inserts the checks which may be as simple as inserting a for loop with conditional clause.

**T2. Add Audit Interceptor** [194]. How can you make it easy to add and change auditing events?

A developer specifies where to intercept data to audit, and how to create audit data. The transformation adds a component that intercepts requests and responses, and creates audit events.

**T3. Add Authentication Enforcer** [194, 184, 101]. How can you ensure that only a set of permitted users enter a system and each user has a separate identity?

A developer specifies the insertion point of an authentication component, source of input data, authentication mechanism (such as username and password, username and certificate, or Kerberos tickets), and outcome of a successful or a failed authentication. The transformation composes classes of a security frame-
work (JAAS, .NET etc) to create a component with the specified authentication mechanism, and delegates authentication requests from the insertion point.

**T4. Add Authorization Enforcer** [194, 184]. How can a system use a subject’s credentials to determine whether a user is allowed to access a resource?

A developer specifies the insertion point of an authorization component, the subject of authorization passed to the component, and the authorization policy. The transformation composes classes of a security framework (JAAS, .NET etc) to create an authorization enforcer component, and delegates access control requests to it.

**T5. Add Password Synchronizer** [194]. How can you ensure that systems sharing user credentials do not have outdated credentials?

A developer identifies the systems that use a password synchronization service. The transformations create the service as a hub between all the components.

**T6. Add Perimeter Filter** [194]. How can you scan and check incoming data for malicious content?

A developer specifies where the input is checked and what are the validation policies. The program transformation adds a new perimeter filter component that applies policies to validate data and delegates all input validation requests to the component.

**T7. Checkpointed System** [21]. How can you ensure that a system can restart gracefully from a crash without losing state information?

A developer specifies where the system checkpoints are and the state information to save at checkpoints. The transformation introduces method calls to save information.

**T8. chroot Jail** [93]. How can you prevent an attacker from corrupting important files?

A developer specifies the constrained environment for a program, including: (1) the location of the chroot directory, (2), the program to be jailed and the programs that communicate with it, (3) the line number of the program where the chroot system call is to be inserted, and (4) the privilege level of the writer and the reader process. The program transformation creates a jail environment for a process and runs it inside a chroot jail.

**T9. Controlled Process Creation** [184]. How can you ensure that privileges cannot be leaked from a parent process to child processes by default?
A developer specifies the parent and child processes and the privileges to transfer. The transformation intercepts a process creation system call and explicitly assigns the privileges to the child process.

**T10. Credential Tokenizer** [194]. How can you ensure that user credentials could be shared by different systems?

A developer specifies the types of tokens to use. The transformation creates a tokenizer that creates and manages the security token by introducing the functionality to encapsulate credentials as a standard token.

**T11. Decorated Filter** [93]. How can you apply multiple input validation policies?

A developer specifies the target input variable and the policies to be applied. The program transformation adds the policies to an input variable by using the Decorator [74] pattern.

**T12. Encryption/Decryption** How can you ensure that an eavesdropper cannot see the content of a message in transit?

A developer specifies the input data buffer, the parameters of the signature algorithm, the key used to encrypt/decrypt a message, and the key encoding algorithm. The transformation composes classes from a security framework (JCA, JCE, .NET etc) to encrypt/decrypt the message.

**T13. Error Message Suppressor** [118, 101]. How can you ensure that attackers cannot use error messages for a reconnaissance attack?

A developer specifies the interception point of error messages (e.g. intercepting all write messages to stderr) and the policies applied to suppress error messages. The transformation adds an interceptor component that applies the suppression policies.

**T14. Exception Shielding** [101]. How can you preserve application behavior when rectified user inputs cause an unexpected state?

A developer specifies exception type, insertion point and data obfuscation policy. The transformation inserts exceptions to catch and process an error condition; at the same time, it obfuscates the error message produced by the exception so that the system does not reveal internal information.

**T15. Explicit Type Enforcement.** How can you secure a program in which arithmetic operations have operands with different types and the end result may contain an unexpected value?

A developer specifies the target input variable. The program transformation checks whether the input variable has been used in an unsafe context and explicitly casts the type of variables in an arithmetic operation, or it changes the type of the variables used in an operation.
**T16. Fuzzing.** How can you test whether a system can survive attackers’ attempt to crash it by feeding garbage inputs?

A developer specifies the program insertion points where fuzzing tests are to be applied as well as the policy used to generate inputs. The transformation automatically adds a component that generates random inputs based on user specification.

**T17. Guarded Object.** How can you design a system that can take access control decisions locally?

A developer specifies a resource to be protected and the associated permissions. The transformation creates a guarded object that encapsulates a resource with the permissions associated with the resource.

**T18. Least Privilege [210].** How can you analyze a program to make sure that it runs with the lowest privilege required to perform all its tasks?

A developer specifies the program to be analyzed. The transformation automatically analyzes a program and identifies the lowest privilege level required; then it lowers the privilege level of the program.

**T19. Message Caching [101].** How can you prevent replay attacks on a system?

A developer specifies the policy enforcement points at system perimeter where incoming messages are accepted, and how to uniquely identify incoming messages. The transformation implements a message cache with replay detection mechanism; it drops all the packets that have been replayed.

**T20. Message Digest Creation.** How can you ensure that an active attacker cannot tamper the content of a message in transit?

A developer specifies the input data buffer, and the message digest algorithm to be used. The transformation composes classes from a security framework (JCA, JCE, .NET etc) to calculate message digest.

**T21. Message Intercepting Gateway [194].** How can you scan and check incoming data for malicious content?

A developer specifies the location of a WSDL file for a service and what policies to apply. The program transformation creates a wrapper component from the WSDL specification; the wrapper component acts as a proxy and delegates the requests to the services, but only after validating the input with a set of filters.

**T22. Partitioning [210].** How can you design a system that constrains a security failure in one part from affecting another part?

A developer specifies how functions are distributed among partitions, what the partition interfaces are, how the partitions communicate, and what privilege levels they have. The transformation distributes the
functions based on the specification, and modifies the privilege levels according to the specification.

**T23. Policy Enforcement Point** [220] [194]. How can you centralize authentication, authorization and input validation at the system entry point so that managing the security operations are easier?

A developer specifies the system entry point and the authentication, authorization and input validation policies. The transformation creates a policy enforcement component and components for authentication, authorization and input validation; it delegates incoming requests to the policy enforcement point component.

**T24. Randomization** [118]. How can you protect applications from an exploit that has been discovered in another instance of the same application?

A developer specifies an artifact of the program (e.g. a data variable, instruction set etc.) that he wants to make distinct. The transformation randomizes the artifact in order to diversify the system so that it survives common vulnerabilities.

**T25. Replicated Component** [21]. How can you ensure that services remain available even after some of them fail?

A developer identifies the network points where replication is required and describes the workload balancing policy. The transformation creates replicas at the network points and creates a workload management proxy at those points; the proxy switches to another service when a service fails.

**T26. Resource Management** [93]. How can you prevent an attacker from launching an application level denial of service attack by exploiting program loopholes and crashing the application?

A developer identifies the program variables that represent resources, and the management policy, e.g. free after use, or check before access etc. The transformation adds resource management checks at the program points.

**T27. Safe Library Replacement** [93]. How can you secure a program that uses unsafe functions leading to possible injection attacks?

For each unsafe function, a developer specifies the alternative safe function and the library that includes the function. The program transformation finds all functions that need to be replaced, replaces unsafe functions with suitable alternatives in all source files, and adds information about the new library to configuration files so that the new program compiles.
**T28. Safe Type Replacement** [93]. How can you secure a program that uses unsafe functions leading to possible injection attacks?

For each unsafe function, a developer specifies the alternative safe function and the safe library. The program transformation finds all functions that need to be replaced and replaces the datatype of function parameters so that the safe function could be introduced; then it replaces unsafe functions with suitable alternatives.

**T29. Secure Logger** [194]. How can you ensure that the causes of an error or an exploit in a system could be detected after the fact?

A developer specifies the messages to log, and policies to retain confidentiality and integrity. The transformation adds a logging component that encrypts and signs logged data; the status messages are delegated to the logging component.

**T30. Secure Resource Pooling** [93]. How can you prevent attackers from exploiting the long lifetime of daemon processes?

A developer specifies the parameters that determine the lifetime of worker processes in a resource pool. The transformation creates a monitor process for the resource pool that kills processes in the pool and replenishes it with new processes.

**T31. Secure Session Object** [194]. How can you ensure that clients do not have to authenticate for every request that they make?

A developer specifies what to put inside a session object, e.g. credentials, roles and privileges. The transformation creates and manages a session object; the session object is shared between a server and a client.

**T32. Signature Generation** How can you verify the sender of a transmitted message?

A developer specifies the input data buffer, the parameters of the signature algorithm, the key used to sign a message, and the key encoding algorithm. The transformation compose classes from a security framework (JCA, JCE, .NET etc) to generate message signature.

**T33. Single Access Point** [220] [184]. How can you secure a system from outside intrusion?

A developer specifies the access points. The program transformation extracts access functions in separate gatekeeper components and introduces a thin unification layer between gatekeeper component(s) and internal components; the layer is a created by making a Façade [74] in the object-oriented world or by
introducing a wrapper component that has the same API.

**T34. Single Sign On Delegate** [194]. How can you ensure that clients do not have to authenticate for every request that they make?

A developer describes the services that share single sign on mechanism. The transformation automatically creates an SSO delegator with default services; the SSO Delegator resides in the middle tier between the clients and the identity management service components, and delegates the service request to remote service components.

**T35. Standby** [21]. How can you ensure that a system gracefully recovers from a crash?

A developer specifies components for which standby components are required and points out what state information of the original component are to be kept. The transformation creates replicas and adds a monitor that transfers control to a standby component when a component fails.

**T36. Tandem Component** [21]. How can you detect and isolate a failing component, so that an independent failure of one component will not cause a system failure?

A developer identifies the component to replicate, the network points where replication is required, the state information of the original component that are to be kept, and how state informations are handled by a comparator component. The transformation creates replicas at the network point and comparator components to check results; the comparator component keeps state information of the replicas and if the states do not match, the comparator concludes that a fault has occurred in one of the components and takes corrective action.

**T37. Unique Location for Each Write Request** [93]. How can you protect shared files from data corruption when file locking mechanism is unreliable?

A developer specifies the sections of a program that accesses a shared file in the reader and writer process, new file creation policy, and information about the shared file. The transformation modifies the write request so that a new file is created for each write request.
References


[34] Bugtraq Vulnerabilities List.


[74] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Addison-Wesley, 1995.


241


242


