Security Oriented Program Transformations
(Or How to Add Security on Demand)

Munawar Hafiz
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
mhafiz@uiuc.edu

Abstract

Security requirements change. Many systems fail to cope with the changing requirements because it is hard to redesign. I show that security can be added by applying program transformations. This improves traditional security engineering approaches and keeps software secure in the face of new security threats.

Categories and Subject Descriptors: D.1.2 [Automatic Programming]: Program transformation; D.2.11 [Software Engineering]: Software Architectures—Patterns

General Terms: Design, Security.

1. Introduction

Security is architectural; it is a property of the entire system, not one part of it. Security cannot be added to a system by adding a module, but it can be added in other ways.

I am interested in automated transformations that change programs to eliminate security threats; they can be source to source or 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.

2. Motivation

The use of program transformations revolutionizes traditional security engineering approaches. Traditional approach relies on designing security from the ground up and writing patches to fix new vulnerabilities. It recommends careful upfront design because "security cannot be added on, it must be designed from the beginning" [1]. However, software architects cannot redesign every time security requirements change. On the other hand, a patch fixes a vulnerability at a fixed set of points; it does not globally remove the vulnerability. Moreover, patches are written hastily when a new vulnerability is reported. This often leads to buggy patches.

Imagine a world where general purpose transformation tools are available for developers. These would mostly be written by a new brand of *program transformation tool vendors*, but some transformations might require custom tools.

Software architects could use the tools to *add security on demand*; at the same time they would be able to feasibly test competing security solutions. On the other hand, software maintainers could respond to a vulnerability report by using a tool to create a security-upgraded program and distribute it as a patch. I call these *program transformation patches* to distinguish from ad hoc security patches. If a tool is not available, a software vendor would release a security patch same as before. At the same time, tool vendors would start new tool development. Finally, a software vendor would apply the tool and distribute a program transformation patch.

The new methodology overcomes the deficiencies of traditional approaches. Changing software, both during development and maintenance, would be more feasible. Instead of redesigning, software maintainers would be able to add security solutions. A program transformation patch would not be limited in scope; it would remove all instances of a security vulnerability. Unlike patch writers, developers of program transformation tools have less strict deadlines. Their tools and the resultant patches could be tested thoroughly for the absence of regression errors.

3. Program Transformations For Security

A *security oriented program transformation* takes an input and a specification and automatically generates an output augmented with the specified security protection.

My research attempts to answer three questions.

(1) Which security solutions can be described as program transformations? Which can not?
(2) How are the program transformations automated?
(3) What are the issues that affect how program transformations are applied, and how they are composed?

4. Results

4.1 Comprehensive List of Security Solutions

Which security solutions can be automated? I have approached this problem from two perspectives: analyzing the problem domain and the solution domain.

Security problems are captured by threat models. I have analyzed threat models of many types of software and read vulnerability trend reports. This analysis reveals that security problems can be grouped into broad classes, e.g. injection attacks, attacks on authentication etc. I have focused on solutions for these classes of security problems.

To get a list of solutions, I have studied security patterns. I am maintaining a comprehensive list containing ninety patterns described in several books, catalogs and papers. I

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

have classified the patterns into disjoint groups of similar patterns [3]. The classification shows that patterns in some groups cannot be automated. This narrows down the initial list to forty four security patterns that can be automated.

Which patterns cannot be automated? Patterns that describe asset evaluation and threat modeling require manual presence and cannot be automated. Some solutions are very context-specific. It is possible to automatically set up a firewall, but the automation has little value because every system is different and requires a customized firewall.

On the other hand, a security solution such as partitioning a monolithic process into multiple processes can be automated. The next section describes some examples.

4.2 Mechanism of Program Transformations

I have so far described twenty one security solutions as program transformations. Most of these descriptions are short; some only survey attempts to automate transformations, while some are supported by proof-of-concept implementations. Developing a commercial quality tool is not my goal, nor is my goal to cover a small group of program transformations. I want to cover the breadth by describing automation mechanism for most security solutions.

Some program transformations that I have described are partitioning a monolithic process into multiple processes, running a process in a constrained environment, checking for buffer bounds for every buffer write operation, and rectifying inputs to prevent various types of injection attacks. Partitioning has been automated many times; I have reviewed the partitioning tools. I have provided a detailed description of how to transform a program to run inside a chroot jail, which is one way of running a process in a constrained environment. I have written a proof-of-concept tool for replacing an unsafe I/O function with a safe function, and have applied the tool over existing programs to remove known buffer overflow vulnerabilities. I have also written an Eclipse plugin for a new refactoring that applies policies to rectify inputs.

A security pattern provides a general description of how a security solution should be like. It does not provide details of how to implement the solution, let alone hint the automation steps. For each of the forty four automation candidates, one has to map the general solution to concrete implementation mechanisms and formulate steps to automate them. For example, the general solution to prevent buffer overflow is to check for buffer bounds for every write operation, as described by 'safe data buffer' [2] pattern. There are many techniques to prevent buffer overflow attacks following the general solution; the most prominent is using safe functions that prevent write operations beyond buffer bounds. Replacing all instances of unsafe functions in a C/C++ program with safe functions prevents various types of buffer overflow. I have written a proof-of-concept Perl script that replaces instances of unsafe $strcpy$ and $strcat$ functions with safe $g\_strlcpy$ and $g\_strlcat$ functions from glib library. The script includes the library information in makefiles and configuration files so that the modified program compiles correctly. I have applied the script on gnu viewer and zziplib, two open source programs with recently reported buffer overflow vulnerabilities. On both occasions, the output program does not have any buffer overflow vulnerabilities. Other examples are replacing $gets$ with $fgets$, $memcpy$ with $memcpy\_s$, $getenv$ with $getenv\_s$ etc.

4.3 Applying Program Transformations

Security oriented program transformations raise issues that affect how transformations are automated, how they are applied and how they are composed.

- *Detail of Specification.* The detail of manual specification varies. An architectural transformation such as partitioning would require more specification than a transformation that affects each part of the program (e.g. replacing string libraries).
- *Specific Context.* Each program transformation has a specific context; it is applicable to a specific program representation. Different programming languages would have different sets of program transformations. For example, a buffer overflow prevention transformation is relevant for a C-like language only.
- *Defense in Depth.* A system should have multiple layers of security tactics, instead of a single security strategy [4]. Composing multiple transformations is hard. Some transformations would be orthogonal, while others might be useful only if applied in a specific way. A transformation to partition a program should be applied before a transformation to run a program inside a chroot jail, otherwise it would make little sense. I have created a pattern language for security patterns [3] that guides the order of composing multiple transformations.

5. Broader Impact

My work would modify security engineering process and activities of its role-players. Achieving security on demand depends partly on research, partly on training application developers, but mostly on the will of tool vendors. Building a tool for a single application is usually not cost effective, so it will need to be done by platform and tool vendors, who can amortize the cost over many users. Once there are good program transformations, application developers will be eager to use them. Then they can make their systems more secure to respond to new threats.

Acknowledgement

Thanks to my advisors Ralph Johnson, Carl Gunter, Sam Kamin and Darko Marinov of UIUC and David Garlan of CMU, and to anonymous reviewers.

References

[1] James P. Anderson. Computer security technology planning study. Technical report, ESD-TR-73-51, Oct 1972.

[2] Munawar Hafiz and Ralph Johnson. Evolution of the MTA architecture: The impact of security. *To be published in Software—Practice and Experience*.

[3] Munawar Hafiz, Paul Adamczyk, and Ralph E. Johnson. Organizing security patterns. *IEEE Software*, 24(4):52–60, July/August 2007.

[4] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems The Right Way*. Addison-Wesley, 2002.