Security-oriented Program Transformations

[Extended Abstract]

Munawar Hafiz, Ralph E. Johnson
University of Illinois at Urbana-Champaign
mhafiz, johnson@cs.uiuc.edu

Abstract

Security experts generally believe that, “security cannot be added on, it must be designed from the beginning” [1]. This is because the typical way of improving system security by patches is ad hoc and has not produced good results. My work 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. I have been maintaining a catalog of security-oriented program transformations; so far the catalog contains forty two transformations. These transformations improve the traditional approaches of security engineering and keep software secure in the face of new security threats.

Categories and Subject Descriptors D.1.2 [Automatic Programming]: Program transformation; D.2.9 [Software Management]: Software maintenance; D.2.11 [Software Architectures]: Patterns.

General Terms Security

Keywords Program Transformation, Security.

Introduction

Security is a property of the entire system, not one part of it. Security is usually not added to a system by adding a module, but it can be added in other ways. This paper shows that it is possible to improve the security of a system by applying program transformations to the existing code base.

When a system starts to show security vulnerabilities, its owners are faced with the choice of replacing it or fixing it. Replacing the system can be expensive and risky, so usually the choice is to fix it. But the most common approach to fixing security vulnerabilities is to patch them, i.e. to fix each instance of a vulnerability separately. Users see a constant stream of patches, leading security experts to claim that it is not possible to add security to existing systems [1] [17] or at least it is economically infeasible to do so [3]. However, this is surely an exaggeration. Even if it is more expensive to retrofit security to a system than to design it in from the beginning, it will be even more expensive to rewrite a system from scratch each time a new type of vulnerability appears.

Any large software system is constructed incrementally; one feature is added before another, and components are split and recombined. There is no obvious reason why techniques that are used on new system cannot be retrofitted to an existing system, especially if these techniques can be automated. Verifying this observation requires making a list of all techniques for making systems secure and checking that each of them can be retrofitted to an existing system.

We have studied the available security solutions and collected them in a catalog of forty two security-oriented program transformations. The list was created by surveying several vulnerability trend reports (e.g. [5, 22]) to find relevant security problems, and using our catalog of security patterns [12] to identify solutions for those problems. Our previous works defined security-oriented program transformations [10], and showed how a collection of program transformations could be applied to eradicate data injection attacks [15], and systematically introduce perimeter security [12]. The focus of this paper is not on these program transformations, or any program transformation in particular. Instead, the focus is on the entire catalog of forty two program transformations. It also describes ways of categorizing these transformations, our first attempts at validating the catalog, and open research problems. More detailed description of the list and the security-oriented program transformations in the list is available as a technical report [14].

Security-oriented Program Transformations

A security-oriented program transformation maps programs to security-augmented programs, i.e. it introduces a security solution. In some cases, these transformations are easy to perform manually. In others, they can be automated. Some program transformations require making significant changes
to the system, for instance adding a new security component. Other transformations are small but ubiquitous, for example replacing unsafe function calls with safe calls (Safe Library Replacement transformation). Although a developer should choose the replacement functions, such a transformation should also be applied by a tool, rather than manually, to avoid developer mistakes. Yet other transformations can be performed with already existing tools.

One of the biggest issues of security-oriented program transformations is how general a transformation needs to be. No transformation will work with every program; they usually expect a certain programming language or a certain platform and apply to programs written in that language or platform in general. Different programming languages and platforms would have different sets of transformations. On the other hand, a patch program uses diff outputs to transform a program, but it is designed to work with a single version of a program. It is too specialized to be a security-oriented program transformation.

Deriving the Catalog

Our catalog of forty two security-oriented program transformations derives from earlier work on security patterns. We have been maintaining a comprehensive catalog of security patterns [13]. The catalog lists ninety one patterns that appear in many books, catalogs and papers on security patterns.

To produce a catalog of security-oriented program transformations from a security pattern catalog, we performed three steps. The first step is to select the candidate security patterns that could be described as program transformations. Not all security patterns are about code and software design. There are security patterns for asset evaluation, risk assessment, and threat modeling [21] that describe a process, and could not be described as a program transformation of software artifacts. Also, security patterns that describe security principles such as Defense in Depth [23] do not have any corresponding program transformations. Removing all these security patterns from the original list of ninety one, we have selected thirty six patterns that could be described as program transformations.

The second step is defining the mechanics of the transformation. 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 [11] security pattern provides a solution to prevent buffer overflow attacks. Buffer overflow occurs because an unsafe language such as C do 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. This pattern have been implemented by a number of safe string libraries (e.g. strlcat and strlcpy [18], libmib library [7], libsafe library [2] etc) and safe data types [19, 11]. We have described the solution as a Safe Library Replacement transformation. The transformation finds all instances of an unsafe library in a program and replaces each instance with a corresponding safe library.

The third step is finding how to automate a program transformation. Not all program transformations can be automated. But if a program transformations could be automated, making it a part of automated tools will make it easier for the developers to apply them. Some program transformations could be completely automated, but most transformations require some manual intervention. Building automated program transformation tools would benefit if the tools make structural changes only, similar to automated refactoring tools. We refer to this as the separation of policy (manual specification) from mechanism (structural change).

Organizing the Catalog

Our catalog is organized on three orthogonal criteria, impact of transformation, vulnerability fixed by transformation, and type of transformation. Combining these criteria gives a three-dimensional categorization model.

Impact of transformation denotes the impact of applying a transformation on a codebase. We have identified four categories of impact that a transformation has on a codebase.

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.

The goal of a program transformation is to fix a single (or maybe more) type of security vulnerability. Vulnerability trend reports such as those from OWASP [20], MITRE [5] and US-CERT [22] identifies most important security vulnerabilities that affect modern systems. Our classification has five distinct types of vulnerabilities that are the most prominent: 1) Unvalidated Input, 2) Broken Access Control, 3) Broken Authentication, 4) Improper Error Handling, and 5) Denial of Service.

Another criteria for grouping transformations is based on the type of change they introduce to the system. We have identified four types of code changes. They are:
1. **Add Component.** Instantiate a component that adds a new security functionality and plug it in the existing component.

2. **Limit.** Set a limit on the resources and check it before a resource is used.

3. **Mutate data.** Change encoding of data, validate or rectify data to remove data related security vulnerabilities.

4. **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.

   We will describe in this paper one group of security-oriented program transformations that prevent the unvalidated input vulnerability. Insufficiently validating input data is the root cause of various types of injection attacks, which is one of the most often exploited security vulnerabilities. Table 1 describes the transformations in this group.

<table>
<thead>
<tr>
<th>Add component</th>
<th>Limit</th>
<th>Mutate data</th>
<th>Distribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Component</td>
<td>Add</td>
<td>Add</td>
<td>Add</td>
</tr>
<tr>
<td>T17. Add Perimeter Filter</td>
<td>T23. chroot Jail</td>
<td>T20. Fuzzing</td>
<td>T42. Unique Location for Each Write Request</td>
</tr>
<tr>
<td>Large change</td>
<td>Change beyond boundaries</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Security-oriented Program Transformations for preventing Unvalidated Input vulnerabilities

Most transformations in this category are about applying input validation checks to program variables. Each transformation modifies a small amount of code, specifically, the code where input data is received.

**Add component.** Since the change has 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. 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.

**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 by analyzing a program and finding its dependencies on static and dynamic libraries, file descriptors, etc. The tool could 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 [9] 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 [8] 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.

### Evaluation

The usefulness of any catalog of solutions depends on how much do the members of the catalog cover a problem area.

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. Any one particular program transformation, such as Add Authentication Enforcer does not solve all authentication problems; it only contributes to the overall security of the system. Nevertheless, transformations in our catalog cover the most relevant security problems faced by today's developers.

We have surveyed the Bugtraq [4] list in the first week of September 2008. 54% of the vulnerabilities in that week were some form of input validation problems, 13% were related with bad authentication and bad access control, 17% were denial of service problems, 1% were error handling problems, and 15% were miscellaneous other problems or combinations of the aforementioned problems. In reality, the partitioning is not mutually exclusive, 33% of the denial of service problems in the survey could be solved by some sort of input validation. Our program transformation catalog contains transformations that could 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. In the remaining two cases, buffer overflow vulnerabilities originate from direct manipulation of pointers. Also, there were 34 SQL injection and 21 cross-site scripting attack instances that could be solved by Decorated Filter transformation. Other problems originating from unvalidated inputs (injection attacks) and improper error handling could also be solved by transformations in our catalog [15]. The authentication and authorization related transformations in the catalog could be used to add new components, or strengthen existing components, that could eradicate most of the problems originating from broken authentication and broken access control.

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. Possibly the easiest way to implement transformations that weave in method calls is to use aspect-orientation [16]. On the other hand, program transformations that rearrange the structure, such as Partitioning, are more like refactorings, and would be hard to implement using aspects. All these transformations can probably be implemented by a code transformation language such as TXL [6] and Stratego/XT [24], or a framework for implementing refactorings. A more philosophical question is whether a domain-specific language would benefit the development of transformation tools.

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 [13] could provide a guideline for the order of composing corresponding program transformations.

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.

Conclusion
Security-oriented program transformations are required to improve the security of legacy systems. 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. Security-oriented program transformations are practical, and are a key to making and keeping our systems secure.

References