Systematically Eradicating Data Injection Attacks using Security-oriented Program Transformations

Munawar Hafiz, Paul Adamczyk, and Ralph Johnson

University of Illinois at Urbana-Champaign
201 N Goodwin Avenue, Urbana, IL 61801, USA
Email: {mhafiz, padamczy, rjohnson}@illinois.edu

Abstract. Injection attacks and their defense require a lot of creativity from attackers and secure system developers. Unfortunately, as attackers rely increasingly on systematic approaches to find and exploit a vulnerability, developers rely on the traditional way of writing ad hoc checks in source code. This paper shows that security engineering to prevent injection attacks need not be ad hoc. It shows that protection can be introduced at different layers of a system by systematically applying general purpose security-oriented program transformations. These program transformations are automated so that they can be applied to new systems at design and implementation stages, and to existing ones during maintenance.

1 Introduction

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, writing patches is an ad hoc, manual activity; developers do not have general purpose tools to prevent these attacks. For example, Bugtraq [7] lists 28 instances of buffer overflow vulnerabilities affecting software from 22 different vendors in the first week of September 2008. 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 function. Despite the common problem and solution, there are no general purpose tools that these developers can use. Moreover, developers work under time pressure and write patches that are very specific to the vulnerability and the affected system. For example, Bugtraq lists that Microsoft Word 2000 has 32 instances of data injection vulnerabilities. Each time developers created a patch to remove the specific vulnerability, only to let the same vulnerability resurface later in another part of the program. The ad hoc nature of security engineering gives an attacker an advantage.

This paper shows that it is possible to think of applying security in terms of general purpose program transformations. Our previous work [14] introduced

the concept of security-oriented program transformations and how using transformations could overcome the problems of traditional approaches of security engineering. This paper describes how to use security-oriented program transformations to protect systems from data injection attacks.

OWASP [28] 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, log injection attack corrupts data in system log files, 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.

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 these data injection vulnerabilities. Most of these 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 are interested in automated, general purpose program transformations that remove security threats from programs. This paper introduces eight program transformations to combat data injection attacks and describes three of them in detail. The first one is an architectural transformation that centralizes input checks at the system entry point. The second is a program-level transformation that applies multiple policies to validate data variables. Our third example, following the defense in depth principle [36], replaces the use of unsafe library functions that cause various injection attacks with safe functions. In all these transformations, developers work at the level of policies. They specify the *policy* (e.g. specify where to add an input validation component, which filters to add to validate input, which unsafe functions to replace), while the *mechanism* is executed automatically by the transformation.

This paper makes the following contributions:

- We present a suite of security-oriented program transformations to counter data injection attacks.
- We show that separating the specification of the policy of securing software systems from the mechanism of making the structural changes makes eliminating security threats more systematic.
- We show with proof-of-concept tools how these security-oriented program transformations could be applied to existing programs.

The next section introduces security-oriented program transformations. Then we describe the security-oriented program transformations that add protection against data injection attacks. We discuss three example program transformations in detail. This is followed by a discussion on the diversity of input validation policies supported by these transformations. Finally, we discuss how general purpose tools can be built to automate the program transformations.

2 Security-oriented Program Transformation

A program transformation is a function that maps programs to programs. A *security-oriented program transformation* maps programs to security-augmented programs, i.e. it introduces a security solution to make programs more secure[1]. We have compiled a list of forty four candidate security solutions [14] that can be described as security-oriented program transformations. For a short description of each of them, see `http://netfiles.uiuc.edu/mhafiz/www/sopt.pdf`.

Program transformations are "general purpose", but no transformation will work with every program. They usually expect a certain programming language or a certain platform, because the exact details are language-specific. By general purpose transformations, we mean that a transformation for a certain programming language should be applicable to all programs written in that language.

Our security-oriented program transformations are "schematic" in the sense that they are structural changes that do not depend on a detailed understanding of the application logic. This property makes them similar to *refactoring* [12], altering the internal structure of code without changing its external behavior. However, security-oriented program transformations are not behavior-preserving the way refactorings are. The concept of behavior preservation in the face of security vulnerabilities is a little different. It should mean that a program preserves the correct behavior and fixes the incorrect behavior that is caused by a security vulnerability. Our program transformations are behavior-preserving when the system is used correctly; they preserve the good path behavior. Only attackers see change in the behavior, because security-oriented program transformations eliminate the source of vulnerabilities the attackers want to exploit.

To use a security-oriented program transformation, a developer must select the *structural* changes, i.e. where to apply a transformation in the code, that can bring about a desired *behavioral* change that eliminates security vulnerabilities, but preserves good path behavior. Usually, a developer supplies these parameters to a program transformation with a manual specification. The transformation automates the boring details by applying the structural changes to all the matching locations in the code. At the minimum, a transformation has to know which part of a program it will work on. But it is common for program transformations to take more parameters, i.e. a specification describing what the output should be. For example, a program transformation that uses policies to validate input variables require that the developer specify the policies. It does not determine by itself which policies to apply, nor does it automatically determine where to apply the policies. It is important to note that the extra parameter should be simpler than either the implementation of the transformation or the program being transformed. We describe the parameters of each program transformation when we present them in the following sections.

Each program transformation is different; so is the implementation strategy of a transformation tool. The strategy depends on the context (language and/or

[1] From this point, we use the terms transformations, program transformations and security-oriented program transformations interchangeably.

platform). This paper focuses on describing the mechanism of each program transformation. We comment on the implementation issues in section 8.

3 Program Transformations to Prevent Injection Attacks

This paper concentrates on eight program transformations from our list of forty four[2] that could be used to prevent various types of data injection attacks. Table 1 lists these transformations. Our description of the mechanics of each transformation has two parts: 1) what a developer needs to specify and 2) what structural changes are made by the transformation.

Suppose Alice, a developer who wants to secure a system from SQL injection attacks, has these transformations available. She does not want to follow the traditional approach of manually writing input validation checks. These checks follow some input validation policies to determine whether an input is valid; some actively rectify an input to make it valid[3]. Program transformations would allow Alice to specify what validation policies to apply, while a transformation would automatically add filters that implement the policies.

Suppose Alice wants to factor out validation policies that are common for all inputs and apply them on data before they are used in the system. She would specify the system entry points and apply an *Add Perimeter Filter* architectural transformation that would create a policy enforcement point to validate all incoming inputs. There are also policies that are specific to each input. Both common and input-specific policies would have to be grouped together and applied to an input variable. A *Decorated Filter* transformation adds a series of filters to an input variable to apply a group of policies.

It might be impossible to completely filter all malicious inputs. In that case, Alice could use safe functions that are designed to robustly handle malicious data. She could apply a *Safe Library Replacement* transformation to replace all instances of unsafe library functions in a program with safe library functions.

These transformations modify user inputs which might lead a program to an invalid state. Alice could apply an *Exception Shielding* transformation to add exceptions to handle the aberrant states. The transformation additionally replaces the error messages with a generic error message to hide internal information.

Other defense mechanisms limit the consequences of an injection attack. To prevent data corruption of important system files, Alice could apply a *chroot Jail* transformation to run a program inside a constrained jail environment. Some data corruption attacks originate when multiple processes write to a system file but do not have a reliable file locking mechanism. Such a system could be transformed with an *Unique Location for each Write Request* transformation that modifies a program to use individual files instead of sharing.

Finally, a system should securely keep log and audit data for forensics. Alice could apply two architectural transformations, *Add Audit Interceptor* and *Secure Logger*, to introduce components that perform these tasks securely and reliably.

[2] The list is available at `http://netfiles.uiuc.edu/mhafiz/www/sopt.pdf`

[3] We use the term validation to mean both active and passive validation.

Name	Problem	Mechanics of Transformation
1. Add Audit Interceptor	How can you make it easy to add and change auditing events?	Developer specifies where to intercept data to audit, and how to create audit data. Transformation adds a component that intercepts requests and responses, and creates audit events.
2. Add Perimeter Filter	How can you enforce input validation policies on incoming data?	Developer specifies where the input is checked and what are validation policies. Transformation adds a policy enforcement component and delegates requests to it.
3. chroot Jail	How can you prevent an attacker from corrupting important files ?	Developer specifies the constrained environment for a program. Transformation creates a jail environment for a process and runs it inside a chroot jail.
4. Decorated Filter	How can you apply multiple input validation policies?	Developer specifies the target input and validation policies. Transformation adds a decorator [13] to the input.
5. Exception Shielding	How can you preserve application behavior when rectified user inputs cause an unexpected state?	Developer specifies exception type and insertion point. Transformation inserts exception, and obfuscates the error message produced by the exception.
6. Safe Library Replace--ment	How can you prevent injection attacks when sanity checks fail to sufficiently validate inputs and the function that uses the inputs are also vulnerable?	Developer specifies the unsafe functions and safe alternatives. Transformation searches and replaces unsafe functions with safe functions.
7. Secure Logger	How can you ensure that system events are logged timely and in a secure manner?	Developer specifies the messages to log, and policies to retain confidentiality and integrity. Transformation adds a logging component that encrypts and signs logged data.
8. Unique Location for each Write Request	How can you prevent data corruption caused by insufficient locking mechanism when multiple processes write to the same file?	Developer specifies the section of a program that writes to a shared file and new file creation policy. Transformation modifies the write request so that a new file is created for each write request.

Table 1. Security-oriented program transformations to prevent injection attacks

Some of our transformations are architectural, while some apply at the program level. Some are programming language or platform specific, e.g. an *Exception Shielding* transformation applies to programs written in languages that support exceptions. Some program transformations apply to the source code, while some apply to binaries. Some transformations are less conventional. For example, a *chroot Jail* transformation makes very few changes to a program's source code or binary. Instead, it transforms the runtime environment of a program to create a jail environment and then run the program inside the jail.

The next three sections describe three example security-oriented program transformations: *Add Perimeter Filter*, *Decorated Filter* and *Safe Library Replacement*. To describe the transformations, we have augmented Fowler's format [12] for describing refactorings.

4 Add Perimeter Filter Transformation

You want to scan and check incoming data for malicious content before it is used in the system.

Add a policy enforcement point at the system entry point to validate data.

4.1 Motivation

For every application, there are some policies that are common for all inputs. Writing checks for these policies for every input variable duplicates a lot of code. One way of eliminating duplication is moving the checks at the application entry point inside a policy enforcement point component.

However, there are policies that are either input-specific or require knowledge of the business logic; these policies cannot be factored out.

4.2 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.
- Program has a few entry points. Before applying this transformation, a developer could apply a *Single Access Point* transformation[4] to minimize the number of entry points. A *Single Access Point* transformation makes a Façade [13] in the object-oriented world or introduces a wrapper component that has the same API.

4.3 Mechanics

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

The program transformation adds a secure base action component [32] that acts as a policy enforcement point [3]. A secure base action component centralizes authentication, authorization and input validation functionality; here we concentrate on input validation. The transformation creates a policy validation component [32] that contains the filters implementing input validation policies. 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.

4.4 Example

Figure 1 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 [1]. A developer also specifies a list of policies.

The transformation creates the `SecureBaseAction` class as a policy enforcement point and the `InterceptingFilter` class that contains the `Filter`s. User inputs are passed to `SecureBaseAction`, which validates by passing inputs through a series of filters.

[4] Part of our larger catalog of forty four transformations.

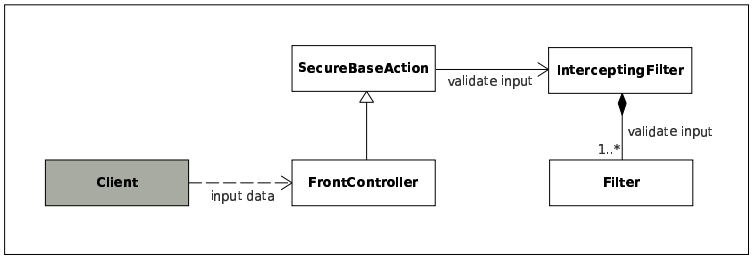

Fig. 1. Applying *Add Perimeter Filter* transformation

4.5 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 [12] for which tools are already available in many programming platforms.

The part that is harder to automate is creating filters from user specification. A 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. Section 7 discusses the various types of policies that could be applied.

5 Decorated Filter Transformation

You want to apply multiple policies to an input variable.

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

5.1 Motivation

Programmers determine a comfort zone [29] of inputs and write checks to prevent inputs outside the zone. Mistakes in writing checks is very common. In the first week of September 2008, Bugtraq lists 34 SQL injection and 21 XSS attack instances that could 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.

5.2 Preconditions

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

- The program has injection vulnerabilities originating from unsafe inputs. Inputs are incompletely or incorrectly checked.
- There are attack patterns that can be used as a basis to implement the validation policies.
- Rectification policies do not transform valid inputs.

5.3 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 [13]. Figure 2 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.

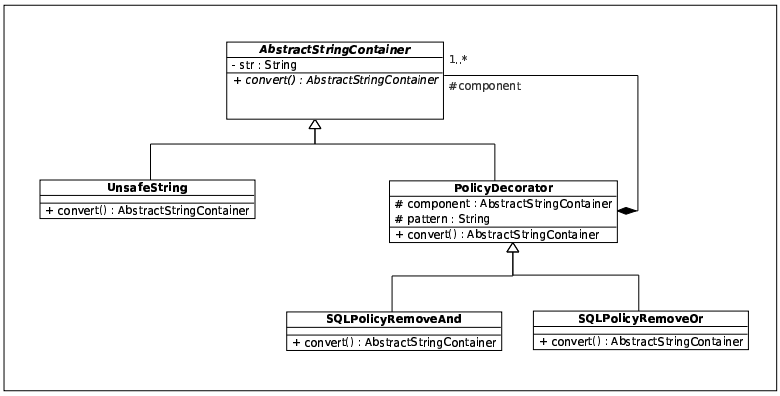

Fig. 2. Class diagram describing policies to apply on an unsafe string

5.4 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 with 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.

```
1  import java.sql.*;
2
3  public class DBConnect {
4    ...
5    public void showData() {
6      ...
7      String username = stdin.readLine();
8      ...
9      try {
10       stmt = connection.createStatement();
11       resultSet = stmt.executeQuery("select * from users " +
12                 "where username = '" + username + "'");
13         } catch (SQLException e) {
14            e.printStackTrace();
15           }
16     }
17 }
```

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 2).

The resulting code is shown here with the changes highlighted.

```
import java.sql.*;
import model.PolicyDecorator;
import model.UnsafeString;
import model.sqlpolicy.SQLPolicyRemoveOr;
import model.sqlpolicy.SQLPolicyRemoveAnd;

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();
         }
    }
}
```

5.5 Towards a Decorated Filter Tool

Applying filters as decorators is an instance of moving embellishment to decorator [23] refactoring. A tool should additionally contain a library of filters and should be extensible. We discuss the diversity of validation policies in section 7.

6 Safe Library Replacement Transformation

You have a program that uses a function that might 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.

6.1 Motivation

Bugtraq lists 28 buffer overflow vulnerabilities in the first week of September 2008. We could 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 originate from direct manipulation of pointers.

Many sophisticated static [35, 38] and dynamic [9, 18] 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 [19, 25], while other functions dynamically resize the destination buffer [24, 21].

Usually, programmers manually search and replace library functions. We asked the developers of the top ten most active projects of all time in sourceforge.net [31] 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 350 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.

6.2 Preconditions

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

- Program uses an unsafe function for which safe alternatives are available.
- Filters in a program fail to sufficiently validate input.

6.3 Mechanics

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 2 lists some unsafe functions that cause buffer overflow attacks in C/C++ and their safe alternatives. Safe alternatives exist for other unsafe functions, e.g. `gets`, `memcpy`, `getenv`, `memmove`, `scanf`, `printf`, etc.

Unsafe string functions	Safe string functions

strcpy(3), strncpy (3) - Copy string
`char *strcpy (char *dst, const char *src);`
`char *strncpy (char *dst, const char *src, size_t num);`
	g_strlcpy from glib library [25],
	astrcpy, astrn0cpy from libmib library [11],
	strcpy_s from ISO/IEC 24731 [20]
	`gsize g_strlcpy (gchar *dst, const gchar *src, gsize dst_size);`
	`char *astrcpy (char **dst_address, const char *src);`
	`char *astrn0cpy (char **dst_address, const char *src, int num);`

strcat(3), strncat(3) - Concatenate string
`char *strcat (char *dst, const char *src);`
`char *strncat (char *dst, const char *src, size_t num);`
	g_strlcat from glib library [25],
	astrcat, astrn0cat from libmib library [11],
	strcpy_s from ISO/IEC 24731 [20]
	`gsize g_strlcat (gchar *dst, const gchar *src, gsize dst_size);`
	`char *astrcat (char **dst_address, const char *src);`
	`char *astrn0cat (char **dst_address, const char *src, int num);`

Table 2. Some unsafe functions and their safe alternatives

6.4 Example

We have written a proof-of-concept Perl script to search and replace `strcpy` and `strcat` functions and applied it to two open source C programs.

Details of the script. The script transforms three types of files.

- *Source Code.* It replaces `strcpy` and `strcat` functions with `g_strlcpy` and `g_strlcat` functions from the `glib 2.0` library. It uses `malloc_usable_size` and `sizeof` functions to calculate the size of heap buffers and stack buffers.
- *Makefiles.* The modified program has to compile correctly. The quickest fix is to include the library during link time. The script modifies *Makefile*s by looking for the pattern `gcc` as a single word and replacing it with `gcc` `` `pkg-config --libs glib-2.0` ``.
- *Configuration Files.* It transforms *config.status* file following the same pattern used for *Makefile*s.

Case studies. We have used the Perl script on two open source C programs: a pdf/ps file viewer (`gv`) and a compression library (`zziplib`). These programs have recent buffer overflow exploits [5, 6], and the exploit codes are available.

`gv` version 3.6.2 has 46 C files with 27000 lines of code. There are 37 `strcpy` and 51 `strcat` instances in the source code. Our script replaces 86 of these 88 instances. Patterns used in our script are not sophisticated enough to replace the two remaining cases; we manually changed those functions. The script also changes 4 lines in the configuration file.

`zziplib` version 0.13.47 has 7346 lines of code in 33 C files. The script changes all 5 instances of `strcpy` and `strcat`. Also, it changes the *Makefile* in 15 places.

In both cases, the resultant programs do not have buffer overflow vulnerability. They compile correctly, pass all tests, and exhibit the same behavior.

6.5 Towards a Safe Library Replacement Tool

Our proof-of-concept tool performs string matching on source code. A commercial quality tool must use a more traditional design with more robust static and dynamic analysis, perhaps a special purpose transformation language such as TXL [8], Stratego/XT [37] and CIL [27]. An approach in this direction is the Gemini tool [10]. 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.

Many safe alternatives exist for each unsafe function. Each safe function in Table 2 uses the same source and destination buffer parameters as the unsafe function; a size parameter is added in most cases. Another approach is to use functions with parameters of a new type [26, 15]. The string library in `qmail` uses a new data structure named `stralloc` [15] to keep length information. To

use these functions, a program transformation replaces all instances of string with the new data structure as well as replacing the unsafe functions.

Safe library replacement tools are not limited to preventing buffer overflows. SQL injection attacks can be prevented by replacing all instances of string concatenation based SQL queries with SQL `PreparedStatement` [34]. Another instance of a safe function is Magicquotes or `addslashes` functions in PHP that automatically escape quotes to prevent SQL injection.

7 Policies for Preventing Attacks on Data

Add Perimeter Filter and *Decorated Filter* transformations apply various input validation policies to input variables. The policies differ for different injection attacks. Table 3 lists some attacks and corresponding input validation policies.

The goal of all injections is to run code in place of data; hence most policies remove/replace keywords and special characters. Attackers try to bypass the checks by encoding inputs. In response, there are some checks that decode inputs and convert them to a canonical format. In Table 3, policies 3 and 9 do so.

Another type of policy augments input with metadata. Such policies are usually applied at the system entry point to augment input and then use it for taint based injection attack detection. WASP [17] augments a Java string with a `MetaString` class to detect SQL injection; SQLCheck [33] adds marks to the beginning and end of a Java string to detect various types of command injection; Pixy [22] adds metadata to PHP strings to detect cross site scripting. These tools apply policies to data twice. First they apply a policy to add metadata to an input at system entry point, and then 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 [16] statically builds an NFA to model SQL statements, CANDID [2] models SQL statements with a parse tree, while SQLRand [4] randomizes SQL keywords in a query to detect the mixing of control and data channels. 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 could 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 that they do 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 refactorings. 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 [30]. In contrast, a security-oriented program transformation needs to additionally know the artifacts that make the behav-

Type of Injection Attack	Example Policy	Description of Transformation
SQL Injection	1. Remove SQL keyword	Searches input for SQL keyword, e.g. SELECT, AND, OR, UNION etc. and removes the keyword. Before: ' OR '1'='1 After: ' '1'='1
	2. Escape single quote	Searches input for single quote characters and escapes them. Before: ' OR '1'='1 After: '' OR ''1''=''1
	3. Decode hex encoding	Decodes any part of the input that has been encoded in hex. Before: 0x73656c656374 After: SELECT
Direct Static Code Injection	4. Remove commands	Searches input for system commands such as system, rm etc. Before: rm%20-rf%20/ After: %20-rf%20/
LDAP Injection	5. Remove $*$ symbol	Searches input for wildcard character and removes it. Before: $*$ After: (Wildcard symbol removed)
	6. Removes & and |	Searches and removes LDAP and (&) and or (|) character. Before: Alice)(|(password=*)) After: Alice)((password=*))
Log Injection	7. Strip special character	Strips input at a special character. Before: Alice 23:45:27 \n root 23:45:24 After: Alice 23:45:27
Cross Site Scripting	8. Remove <SCRIPT> tag	Searches for <SCRIPT>...< /SCRIPT> and removes it. Before: <SCRIPT>alert('XSS');</SCRIPT> After: (<SCRIPT> tag removed)
	9. Decode UTF-8 encoding	Decodes UTF-8 encoding in code. Before: <IMG SRC=javav ascript :alert('XSS')> After: <IMG SRC="javascript:alert('XSS');">

Table 3. Sample policies for various transformations

ioral change. For example, an *Add Perimeter Filter* transformation needs to know about the library of filters to use, to change program behavior.

The program transformations that we have described are orthogonal, they can be applied in any order. However, some policies must be applied in a particular order. The policies that canonicalize an input are applied before any other policies. Some other policies might keep the order information, for example 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. An automated tool can keep these structural details and use them effectively.

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. An organization might have to modify their policy base or create new policies to cope with requirement changes. The library of policies maintained by a program transformation should be parameterizable and extensible.

8 Tools for Security-oriented Program Transformations

None of the security-oriented program transformations described in this paper require deep analysis of the program being transformed. They share many structural similarities with refactorings and can probably be implemented similarly.

Some of the transformations in this paper insert subroutine calls into a program. These could be implemented using an aspect-oriented programming language. Others rearrange the structure of the program. These are more like refactorings, and would be hard to implement using aspects. *Safe Library Replacement* matches a set of code patterns representing calls to an old library, and replaces them with code that calls a new library. It seems unlikely that this could be implemented using an aspect-oriented language. All these transformations can probably be implemented by a code transformation language such as TXL [8] and Stratego/XT [37], or a framework for implementing refactorings.

All the program transformations are platform dependent, at least to the extent that they depend upon the language of the program being transformed, and perhaps on the operating system (*chroot Jail*) or the libraries (*Safe Library Replacement*). Frameworks for refactoring are also platform dependent, as are most of the languages for writing program transformations.

It is unreasonable to expect researchers to build such 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. This will need to be done by platform and tool vendors, who can amortize the cost over many users.

9 Conclusion

Keeping systems secure requires constantly improving them. Security-oriented program transformations are a repeatable, systematic approach to eradicating system vulnerabilities. This paper focuses on data injection attacks, but the idea can be applied to other areas of vulnerability. Security-oriented program transformations have the potential to allow security engineers to develop policies and let a tool apply those policies. This will blend human creativity with the thoroughness of the computer.

Acknowledgement

Thanks to Baris Aktemur, Farhana Ashraf, Sruthi Bandhakavi, Nicholas Chen, David Garlan, Carl Gunter, Sam Kamin, Darko Marinov, Jeffrey Overbey and Maurice Rabb for providing valuable feedback. We also thank US Department of Energy for their financial support under the grant number DE-FG02-06ER25752.

References

1. D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Pearson Education, 2001.
2. S. Bandhakavi, P. Bisht, P. Madhusudan, and V. Venkatakrishnan. CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 12–24, New York, NY, USA, 2007. ACM.
3. B. Blakley and C. Heath. Security design patterns technical guide–Version 1. Technical report, Open Group(OG), 2004.

4. S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, 2004.
5. Bugtraq ID 20978. gv stack buffer overflow vulnerability.
6. Bugtraq ID 23013. zzipLib zzip_open_shared_io stack buffer overflow vulnerability.
7. Bugtraq Vulnerabilities List. http://www.securityfocus.com/vulnerabilities.
8. J. Cordy, C. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. In *ICCL*, pages 280–285. IEEE, 1988.
9. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Seventh USENIX Security Symposium proceedings: San Antonio, Texas*, 1998.
10. C. Dahn and S. Mancoridis. Using program transformation to secure C programs against buffer overflows. In *WCRE '03*, page 323, Washington DC, USA, 2003. IEEE Comp. Society.
11. F. Cavalier III. Libmib allocated string functions. `http://www.mibsoftware.com/libmib/astring/`.
12. M. Fowler. *Refactoring: Improving The Design of Existing Code*. Addison-Wesley, Jun 1999.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
14. M. Hafiz. Security oriented program transformations (Or how to add security on demand). In *OOPSLA '08: Companion to the 23rd annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, 2008. ACM.
15. M. Hafiz and R. Johnson. Evolution of the MTA architecture: The impact of security. *To be published in Software—Practice and Experience*.
16. W. Halfond and A. Orso. Preventing SQL injection attacks using AMNESIA. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 795–798, New York, NY, USA, 2006. ACM.
17. W. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 175–185, New York, NY, USA, 2006. ACM.
18. E. Haugh and M. Bishop. Testing C programs for buffer overflow vulnerabilities. In *NDSS*. The Internet Society, 2003.
19. International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. Dec 1999.
20. International Organization for Standardization. *ISO/IEC 24731: Specification For Secure C Library Functions*. 2004.
21. ISO/IEC 14882. C++ std::string.
22. N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS '06: Proceedings of the 2006 workshop on Programming Languages and Analysis for Security*, pages 27–36, New York, NY, USA, 2006. ACM.
23. J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.
24. M. Messier and J. Viega. Safe C string library v1.0.3.
25. T. Miller and T. de Raadt. `strlcpy` and `strlcat` — Consistent, safe, string copy and concatenation. In *1999 Usenix Annual Technical Conference, Monterey, California, USA*, 1999.
26. A. Narayanan. Design of a safe string library for C. *Software—Practice and Experience*, 24(6):565–578, 1994.
27. G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.
28. OWASP. Categories of injection attacks, 2008.
29. M. Rinard. Living in the comfort zone. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 611–622, 2007.
30. D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, 1997.
31. SourceForge.net. Most active projects - all time., Feb 2008.
32. C. Steel, R. Nagappan, and R. Lai. *Core Security Patterns : Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management*. Prentice Hall PTR, Oct 2005.
33. Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM.
34. S. Thomas and L. Williams. Using automated fix generation to secure SQL statements. In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*, page 54, Washington, DC, USA, 2007. IEEE Computer Society.
35. J. Viega, J. Bloch, T. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference*. ACM, 2000.
36. J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems The Right Way*. Addison-Wesley, 2002.
37. E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, May 2001.
38. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*. The Internet Society, 2000.