A security oriented program transformation to "add on" policies to prevent injection attacks

Munawar Hafiz, Ralph Johnson
University of Illinois at Urbana-Champaign
(mhafiz, rjohnson)@cs.uiuc.edu

Abstract

Topping the list of the most prominent attacks on applications [6] are various types of injection attacks. Malicious inputs that cause injection attacks are numerous; programmers fail to write checks for all attack patterns. We define a program transformation that allows a programmer to think in terms of rectification policies and automatically add these policies to convert unsafe data inputs to safe inputs. The security oriented program transformation applies to all classes of injection attacks, easing the burden of programmers who would otherwise have to manually write checks.

Categories and Subject Descriptors D.2.10 Software [*Software Engineering*]: Design; D.2.11 Software [*Software Engineering*]: Software Architectures; K.6.5 Computing Milieux [*Management of Computing and Information Systems*]: Security and Protection

General Terms Design, Security

Keywords Injection Attack, Program Transformation, Input Rectification Policy.

1. Introduction

Recent exploration into injection attacks tally eighteen variants [3]. This excludes buffer overflow and format string attacks, both of which technically fall under the same category. Despite the exclusion, classes of injection attacks pose the most severe threat [6] on today's applications.

During secure software design, programmers write checks for screening malicious inputs. But, it is difficult for a programmer to enumerate all attack patterns and manually write checks for these conditions. This difficulty is the reason why injection attacks exist in the first place.

WRT '08 Oct, Nashville

This paper shows that it is possible to add protection against injection attacks by applying 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. Our program transformation is similar to a refactoring. However, it improves the security of systems, which means that it does not preserve all types of behavior. It preserves expected behavior, but should change a system's response to security attacks.

Our program transformation applies the Decorator pattern [2] to decorate variables with policies that transform unsafe inputs to safe inputs. Our approach introduces the concept of *input rectification policies*, aimed for security. The automation allows programmers to concentrate on types of applicable policies instead of worrying on how to manually implement the checks.

We automate the refactoring with a proof-of-concept Java plugin that shows how the refactoring is applied to protect against SQL injection attacks. However, the purpose of this paper is not to describe a program transformation or a tool for program transformations. Instead, it is to show that there are *security-oriented program transformations* that allow security to be added on demand.

We use a small Java program to show the effect of our program transformation. The next section lists the program. Then we describe the *injection attack elimination program transformation*. To describe the transformation, we have augmented the format [1] for describing refactorings. In the heart of our refactoring are rectification policies, each of them relevant to a type of injection attack. We conclude with a discussion on these policies.

2. An Example Insecure System

We illustrate how our program transformation can add security on demand with a small Java program. Class DBConnect contains methods for connecting to a database (connect), querying for data and showing result from the database (showData), closing the database connection (close), and launching the program (main). The database contains a

single table named users, with three fields for storing user id, user name and password. Our focus is on showdata method. It reads input from standard input (lines 18–26), prepares the query and executes it (lines 28–30), and shows the result (lines 31–37).

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStreamReader;
4  import java.sql.*;
5
6  public class DBConnect {
7    private Connection conn = null;
8
9    public void connect() {
10     // Connects to the database
11     }
12
13   public void showdata() {
14     ResultSet rs;
15     Statement st = null;
16     StringBuffer query = new StringBuffer();
17     String username = new String();
18     InputStreamReader isr =
19                     new InputStreamReader(System.in);
20     BufferedReader stdin = new BufferedReader( isr );
21     try {
22       username = stdin.readLine();
23       } catch (IOException e1) {
24         e1.printStackTrace();
25         }
26
27     try {
28       st = conn.createStatement();
29       rs = st.executeQuery("select * from users " +
30                 "where username = '" + username + "'");
31       while (rs.next()){
32         int uid = rs.getInt("userid");
33         String uname = rs.getString("username");
34         String password = rs.getString("password");
35         System.out.println((new Integer(uid)).toString()
36                         + "  " + uname + "  " + password);
37         }
38         rs.close();
39         st.close();
40         } catch (SQLException e) {
41            e.printStackTrace();
42           }
43     }
44
45   public void close(){
46     // Close database connection
47     }
48
49   public static void main(String[] args) {
50     DBConnect dbConnect = new DBConnect();
51     dbConnect.connect();
52     dbConnect.showdata();
53     dbConnect.close();
54     }
55 }
56
```

Suppose the user enters the string "Alice" for input. The resultset consists one row containing information about Alice.

Attacking this program is straightforward. A malicious user enters "' or '1'='1" for input. The resulting query is,

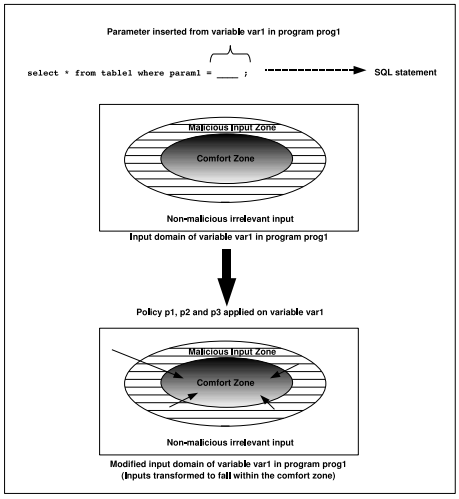
```
select * from users where
         username = '' or '1'='1'
```

Our refactoring allows a programmer to apply rectification policies at design time. When a malicious user enters an unsafe input, the applied policies transform the unsafe input to safe input. The query executes normally and the attack is prevented.

3. Injection Attack Elimination Transformation

You have a program that accepts inputs from users. The programmer has envisioned its input domain, but in practice it accepts inputs outside the pre-conceived domain. Some of these inputs could be malicious; you want to ensure that the program is not vulnerable to attacks originating from the injection of malicious inputs.

Apply policies to input variables that transform unsafe inputs to safe inputs.



3.1 Motivation

There are eighteen different types of injection attacks. They affect various programming environments, e.g. SQL injection attack affects a database manipulation language, LDAP injection attack affects statements of the directory access protocol, cross site scripting injects malicious code through an HTTP payload, log injection attack corrupting data in system log files etc. Although not included in the taxonomy, buffer overflow attack and format string attack are technically injection attacks – during buffer overflow, an attacker injects a malicious input with long length to corrupt the program counter, whereas in the other case, an attacker injects malicious format instructions to corrupt data or reveal system information. Together, injection attacks pose the most insidious threat on modern software.

Programmers determine a comfort zone [5] of inputs and write checks to prevent inputs outside the zone. However, writing input correctness checks for each input is a tedious task, and programmers often make mistakes that allow an attacker to run a program with inputs outside the comfort zone. Besides, correctness tests are not exercised during

the normal execution of the program; therefore, they might receive less attention during testing.

Replacing the manual checking with automated tools allow programmers to concentrate on policies rather than the mechanism of implementing checks.

3.2 Preconditions

This program transformation applies to programs in any object oriented language (that allows accessing and manipulating the abstract syntax tree) running in Unix and Windows environment. However, the concept can be translated to protect programs written in non-object oriented languages. The applicability of the transformation also depends on the policy to be applied. A program with the following characteristics benefits from a buffer overflow elimination transformation.

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

3.3 Mechanism

Refactor code to encapsulate an input variable in the abstract component of a Decorator [2]. Write policies that transform the input to make it safe. Figure 1 describes how a string variable is decorated with policies that remove SQL injection attack vectors.

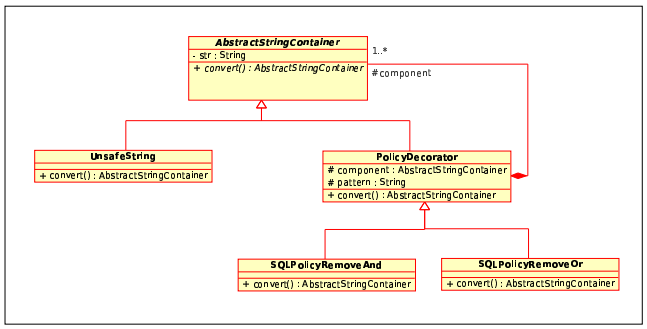


Figure 1. Class diagram describing policies to apply on an unsafe string

Change the source code to import policy classes. Before the input variable is referenced, apply policies to rectify it.

3.4 Example

To illustrate the program transformation, we have written an Eclipse plugin to apply SQL injection prevention policies on Java programs. A programmer has to specify the variable to rectify and the policies.

In the example program, we have applied policies to remove AND and OR statements (from figure 1). The resulting code is shown here with the changes highlighted.

```
import model.PolicyDecorator;
import model.UnsafeString;
import model.sqlpolicy.SQLPolicyRemoveOr;
import model.sqlpolicy.SQLPolicyRemoveAnd;
...
...
  public void showdata() {
    ...
    ...
    UnsafeString username = new UnsafeString();
    ...
    ...
    try {
      s.setStr(stdin.readLine());
      } catch (IOException e1) {
    ...
    ...
    try {
      st = conn.createStatement();
      PolicyDecorator policy =
                  new SQLPolicyRemoveAnd(
                      new SQLPolicyRemoveOr(username));
      rs = st.executeQuery("select * from users " +
                           "where username = '" +
                           policy.convert().getStr()
                           + "'");
      ...
      ...
```

4. Policy Suites for Injection Attacks

Each injection attack is different; it has its own set of prevention policies. Table 1 lists some attacks and their sample input rectification policies.

The goal of all injections is to run code in place of data; hence most policies remove/replace keywords and special characters. Attackers counter this by applying various encoding schemes. Canonicalizing data is very important before applying any removal/replacement policies. Table 1 also list policies for encoding/decoding.

There are many types of encodings that an attacker can incorporate in one input to bypass scans. In this case, selecting all policies is burdensome for a programmer. This problem can be solved by introducing composite policies.

5. Issues

One might argue that this approach tries to enumerate badness [4], which is a common security mistake. In reality, policies do not have to identify bad patterns in data and remove them. Instead, they can describe good properties of data and discard everything that is aberrant.

However, enumerating badness is a common principle embraced by every filter. In this case, our transformation would provide defense in depth by adding one additional layer of checking.

Another issue is what to do with the transformed input that does not make sense. Doctoring malicious inputs would produce such inputs. This causes a program to behave differently, but only from the perspective of attackers. Normal users do not inject malicious inputs, hence the policies never alter their input. An extreme measure is to change the attacker's input to an irrelevant but good input whenever a policy applies to user input. This has two advantages. First,

Type of Injection Attack	Sample Rectification Policy	Description of Transformation
SQL Injection	Remove SQL keyword	Searches input for SQL keyword, e.g. SELECT, AND, OR, UNION etc. and removes the keyword. Input Data: ' OR '1'='1 After Rectification: ' '1'='1
	Strip SQL keyword	Searches input for SQL keyword, e.g. SELECT, AND, OR, UNION etc. and strips the string from that point. Input Data: ' OR '1'='1 After Rectification: '
	Remove/Strip single quote	Searches input for single quote and remove/strip it. Input Data: ' OR '1'='1 After Rectification: OR 1=1 (Remove) After Rectification: (Entire string is stripped)
	Escape single quote	Searches input for single quote characters and escapes them. Searches input for single quote characters and escapes them. After Rectification: '' OR ''1''=''1
	Remove comments	Searches input for SQL comments and removes comments Input Data: ' OR '1'='1 – something else After Rectification: ' OR '1'='1
	Decode hex encoding	Decodes any part of the input that has been encoded in hex. Input Data: 0x73656c656374 After Rectification: SELECT
Direct Static Code Injection	Remove system commands	Searches input for system commands such as system, rm etc. Input Data: rm%20-rf%20/ After Rectification: %20-rf%20/
LDAP Injection	Remove symbol	Searches input for wildcard character and removes it. Input Data: After Rectification: (Wildcard symbol removed)
	Removes & and — characters	Searches input for LDAP and (&) and or (—) character and removes it. Input Data: Alice)(—(password='')) After Rectification: Alice)((password=''))
Log Injection	Strip newline (n) character	Strips string at newline character (or other special characters).. Input data: Alice 23:45:27 n root 23:45:24 After Rectification: Alice 23:45:27
XSS Injection	Remove javascript statement	Searches input for javascript statement and removes it. Input data: ¡IMG SRC=''javascript:alert('XSS');''¿ After Rectification: ¡IMG SRC=''¿
	Remove ¡SCRIPT¿ tag	Searches for ¡SCRIPT¿...¡/SCRIPT¿ and removes it. Input data: ¡SCRIPT¿alert('XSS');¡/SCRIPT¿ After Rectification: (¡SCRIPT¿ tag removed)
	Decode UTF-8 encoding	Decodes UTF-8 encoding in code. Input data: ¡IMG SRC=java vascri pt:ale rt('XS S')¿ After Rectification: ¡IMG SRC=''javascript:alert('XSS');''¿

Table 1. Sample policies for various transformations

it restricts policies to create inputs that are bogus but outside the comfort zone of the program, thus making the program more robust. Second, it postpones checking for further policies and returns, thus improving performance.

6. Conclusion

We have introduced input rectification policies and described a program transformation to apply the policies. Thinking in terms of security policies would motivate in depth studies on policies to prevent various types of injection attacks. This effort would benefit from the studies on attack signature generation. Whether using existing policies or customizing new policies, programmers would be freed from writing boring checks and be able to concentrate on more productive tasks.

References

[1] Martin Fowler. *Refactoring: Improving The Design of Existing Code*. Object Technology Series. Addison-Wesley, June 1999. With contributions by Kent Beck, John Brant, Willima Opdyke, and Don Roberts.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[3] OWASP. Categories of injection attacks, 2008.

[4] Marcus J. Ranum. The six dumbest ideas in computer security. `http://www.ranum.com/security/computer_security/editorials/dumb/`, September 2005.

[5] Martin C. 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, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5.

[6] Andrew van der Stock, Jeff Williams, and Dave Wichers. OWASP Top 10 - The ten most critical web application security vulnerabilities - 2007 update, 2007.