1 Introduction

Applications that accept user input–from forum software to shopping carts–are vulnerable to injection attacks like SQL injection, path traversal and cross site scripting (XSS). We can address these vulnerabilities by detecting and properly handling attacks.

Taint-tracking is aimed at detecting these attacks by keeping whether each string is derived from user input ("tainted"). In such a scheme, all user input is marked tainted, and all strings derived from tainted strings are also marked tainted. Then, the sensitive strings are checked at a sink–such as a SQL query parser–so that unsanitized strings will be rejected.

Tracking taint information at the string level is simple and efficient–it requires a simple `boolean` field to each string–but it results in too many false positives. For example, the SQL query `SELECT * FROM users WHERE firstname='John'` (the underlined portion denotes user input) would be considered dangerous although control characters are not from user input. Thus, it is important to also track which portions of the string are tainted in addition to which strings are tainted.

Erika Chin's work on character-level taint tracking addresses that shortcoming by tracking taint information on a finer granularity. However, it suffers from one major obstacle–performance. Her implementation of taint-tracking using a boolean array incurs a significant overhead on some user requests. We aim to address that by representing taint differently.

Our goal is to optimize the process of propagating, storing and testing taint information for the common case, but maintain the character-level granularity. Our hypothesis of what constitutes the common case is derived from experimenting with and analyzing JForum, a web discussion software based on Java Server Pages. We will evaluate the hypothesis by profiling JForum, and obtain rigorous micro- and macro-benchmarking results. Like Chin's work, this focuses on Java/Tomcat, but the results are relevant to many other server implementations.

2 Our Hypothesis: Motivation for Optimizations

Our hypothesis is that most strings are either untainted, or have a single taint interval, and therefore we should optimize for that. The theoretical basis for the hypothesis rests on the fact that the initial taint properties are very simple (tainted or untainted) and that there are very few ways to increase the number of taint intervals ('complexity').

All user input is fully tainted at the source. All strings extracted from the user request are fully tainted, and all other strings (not derived from tainted strings) are considered untainted. For example, if '`John`' is the value for the parameter `firstname` (from the user request), the entire string is considered tainted.

Concatenating an untainted string with a string with a single taint interval results in a string with a single interval. For example, concatenating '`My name is `' with '`John`' results in '`My name is John`', which still only has a single taint interval.

All substrings of strings with a single taint interval also have at most one taint interval. For example, a substring of '`My name is John`' may be '`My name`' or '`is John`'.

The taint properties can only become complex through concatenation of two tainted strings, or through string replacement (which can be transformed into a call to substring followed by a serial concatenation). For example, concatenating '`My name is John` ' and '`Smith`' will result in a string with complex taint properties ('`My name is John Smith`') while concatenating '`Your suggested username is John`' and '`Smith`' will result in another string with a single taint interval ('`Your suggested username is JohnSmith`').

However, not all strings have such simple taint properties. Such strings are necessarily a result of many concatenations or string replacements (which are uncommon). Therefore, they tend to be longer than untainted strings and strings with single taint intervals.

These strings may have an unbounded number of taint intervals, and must be handled gracefully. For example, an implementation of taint tracking using an interval set as the data structure would quickly succumb to a pathological case like the following:

```
for (String s : user_input)
  list += s + ","; // s is fully tainted but the comma is not
```

Our method of representation must exploit the simplicity of most strings while properly handling less common cases where the string may have an arbitrary number of taint intervals.

3 Representation: TaintSet Data Structure

The taint information is represented in the TaintSet data structure either as an interval–when that is sufficient–or as a bitmap. The dual-representation allows for both performance when possible, and flexibility in general.

The former representation covers a majority of strings, and it is extremely efficient because interval arithmetic is simple. The latter representation covers the strings with more complex taint properties, and it is extremely compact and efficient because doing group operations on bitmaps is faster than manipulating individual boolean array elements.

4 Implementation

All `String`s, `StringBuffer`s and `StringBuilder`s are given an additional field '`taintvalues`' of type `TaintSet` that represents the string's taint properties. This value is `null` if the string is untainted–which is the case for most strings.

We further optimize the storage and performance by adding a constant `TaintSet` object named `allTainted` to the class. This will allow us to avoid instantiating new `TaintSet` objects for all fully-tainted strings, saving both time and space. Thus, `taintvalues = TaintSet.allTainted` if the corresponding string is fully tainted.

For strings with other taint patterns,

5 Outline

1. Introduction
2. Basic Implementation
   (a) Modifications to String Class
   (b) Source Tainting and propagation
3. Profiling
   (a) Measuring taint properties of Strings
      i. Measure the number of taint intervals of strings in calls to string methods
      ii. Histogram based on use (invocation of the listed methods) not existence or creation
      iii. Retrofitted Erika's code because counting distinct intervals in boolean arrays easier
   (b) Measuring frequency of method calls

i. Updated static hash table with each call to a retrofitted String method

ii. Goal: Determine where to focus optimizations

(c) Methodology and Details

i. Track calls to the following methods in String

A. `replace(CharSequence, CharSequence)`

B. `substring(int, int)`

C. `substring(int)`

D. `toLowerCase(Locale)`

E. `toUpperCase(Locale)`

F. `trim()`

G. `concat(String)`

H. `matches(String)`

I. `split(String, int)`

J. `split(String)`

K. `replaceAll(String, String)`

L. `replaceFirst(String, String)`

ii. Static methods in String: Upon first invocation, start a "logger" thread that periodically writes to log files

- No synchronization tools ("synchronized") is used because the cost of a loss of a couple counts is minor

iii. Performed a sequence of actions on JForum running on a server with a retrofitted String class:

A. Visit the home page

B. Click on a forum to view its index page

C. Click on a topic with many posts

D. Click reply

E. Post reply

iv. Inspected log file after each action and kept track of changes

(d) Results

i. Method Calls

Method	A	B	C	D	E
replace	0	0	0	0	0
substring	1825	6765	4619	136	4697
toLowerCase	78	885	734	20	996
toUpperCase	1	12	13	0	10
trim	1533	1111	2	0	144
concat	99	234	0	0	112
matches	0	0	0	0	0
split	0	0	0	0	0
replaceAll	0	0	0	0	0
replaceFirst	0	0	0	0	0

ii. Taint histogram

Taint Interval Counts	A	B	C	D	E
0	3733	11481	7563	156	7577
1	11	11	13	0	76
More than 1	0	0	0	0	0

(e) Future work

i. Track StringBuffer/StringBuilder. Possible explanation: Most concatenation invocations are from Buffer/Builder (For example, the automatic conversion of + to append calls)

ii. Track taint histogram for each method. Perhaps the calls to substring() are not problematic despite the sheer count because those calls typically are on untainted Strings.

4. Testing Methodology and Benchmarks

(a) Test Setup

i. running on gradgrind * get stats *

ii. consistent machine for both micro and macro tests

(b) Microbenchmarking

i. Methodology

A. Testing for steady state since servers will be in such a state

B. Using Benchmark framework * cite source *

- Run framework for x (determine actual number) runs and average over those runs
- Framework automatically runs until steady state per test

- Each run is another instantiation of the VM running a set of tests until steady state/test

C. Tested from command line, switched out VMs to test different versions
- IBM-JAVA
- Basic Implementation (Erika)
- Set Version

D. Test various string class methods
- String Creation
- Concatenation
- Substring
- Trim
- Case Changing
- Replacement

E. Test Various length strings
- 12
- 1024

F. Test Various taint versions for taint-capable VMs
- No taint
- full taint
- 1/2 taint
- 2/3 taint

(c) Benchmarks

i. * show them *

ii. Remarks

A. Explanation for beating ibm-java in certain cases...

(d) Macrobenchmarking

i. Methodology - Latency Testing

A. Measure time from receiving request to finish servicing request

B. Use JForum to simulate webapp with medium reading to writing load

C. Testing POST/GET requests on a pre-populated JForum database

D. Server to Server on LAN * what is capable speed? *

E. Client:
   - Server with a python script being invoked from the command line

F. Server:
   - Server running Tomcat instances with a JForum application (only one running at a time)
   - One version is standard Tomcat6 running IBM-JAVA, other is a modified version to
   - Taint sources running either Erika's VM or the TaintSet VM
   - Mysql instance to handle database requests

ii. Benchmarks

A. * show them *

5. Conclusions/Future Work

(a) Uppercase/lowercase

(b) More comprehensive macrobenchmarks