Efficient Character-level Taint Tracking for Java

Erika Chin
University of California, Berkeley
Berkeley, CA, USA
emc@cs.berkeley.edu

David Wagner
University of California, Berkeley
Berkeley, CA, USA
daw@cs.berkeley.edu

ABSTRACT

Over 80% of web services are vulnerable to attack [4], and much of the danger arises from command injection vulnerabilities. We present an efficient character-level taint tracking system for Java web applications and argue that it can be used to defend against command injection vulnerabilities. Our approach involves modification only to Java library classes and the implementation of the Java servlets framework, so it requires only a one-time modification to the server without any subsequent modifications to a web application's bytecode or access to the web application's source code. This makes it easy to deploy our technique and easy to secure legacy web software. Our preliminary experiments with the JForum web application suggest that character-level taint tracking adds 0–15% runtime overhead.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*Protection Mechanisms*; D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*

General Terms

Security, Performance, Languages

Keywords

dynamic taint tracking, Java, web applications, information flow

1. INTRODUCTION

Web applications provide user services on a global scale without the need to distribute specialized client software. They allow for easy, centralized software maintenance via server-side updates, making it possible to provide useful, inexpensive services to anyone with access to the Internet. Unfortunately, many of these services contain security vulnerabilities, making the use of web services and websites potentially dangerous to a user's security and privacy. It has been observed that over 80% of websites contain security vulnerabilities [4].

Much of the danger arises from data-driven attacks. A malicious user can send a web application malformed input that, if improperly sanitized, can cause the program to behave in an unexpected and undesirable manner. The exploited application may then leak personal information or serve malware to users. Some common attacks in this class include SQL injection, cross-site scripting (XSS), path traversal, response splitting, and shell injection attacks.

One emerging approach to solving this problem is taint tracking. Taint tracking consists of three main steps. The first step is to identify untrusted input at the point that it enters the program and mark that it is untrusted (i.e., tainted). This is called "source identification" or "source tainting." The second step is to propagate taint information as subsequent computation occurs, marking as tainted all data that is derived from an untrusted source. For example, if part of the tainted data is used to create a new variable, that variable also becomes tainted and subsequently tracked as well. Finally, all data going into a sensitive data sink (e.g., a database, response output, or file) is checked, using the taint information to identify potential attacks.

We present an efficient character-level taint tracking system for Java programs. Our approach requires modification only to Java library classes and the Java servlets implementation. Unlike techniques that require bytecode translation on every web application, our approach is a zero-maintenance server-centric approach that does not require any changes to be made to any web application bytecode or source code. It is a one-time server-side modification, making it easy to adopt, deploy, and protect legacy web software. This technique also allows taint tracking to be performed on strings that use Java reflection, unlike bytecode translation techniques.

In this work, we focus on character-level taint tracking: i.e., we track the taint status of each character individually. Instead of considering each string variable as either tainted or untainted, character-level taint propagation tracks the taint status of each character of the string separately, allowing us to determine which portions of the string are tainted. This provides finer-grained information to the sanitization method, and thus contributes to a lower false positive rate. Despite the added analysis, we show that character-level tracking can be performed efficiently. We evaluate performance in the context of Java web services as they often need to interface with untrusted input. Our technique, however,

SWS'09, November 13, 2009, Chicago, Illinois, USA.

can be generalized for any Java program. Our preliminary measurements indicate that character-level tracking adds no more than 15% runtime overhead, suggesting that our approach is efficient and easily deployable.

We are not the first to study character-level taint tracking for web security. There is a great deal of prior work on taint tracking [8, 9, 3, 12, 7, 1, 2, 5, 6], and many authors have explained how character-level taint tracking can be used to build runtime defenses against command injection attacks [8, 9, 12, 10]. Most closely related is work by Halfond et al., who also study character-level taint tracking for Java web applications [6]. Their system works by rewriting the bytecode of the web application. We build upon many of their ideas, but we implement taint tracking by replacing the Java libraries instead of rewriting application bytecode. Because our approach avoids the need for bytecode rewriting, it may be easier to deploy and may be more compatible with legacy code: for instance, we avoid certain technical limitations regarding use of reflection. In addition, we place a special focus on the completeness of our taint propagation policy, and we carefully analyze a number of technical details that were not discussed in that work.

In Section 2, we give an overview of command injection attacks and how character-level taint tracking is effective in protecting against these attacks. In Section 3, we present our Java taint tracking system and our policy decisions. In Section 4, we share our implementation details, and in Section 5, we evaluate JForum's performance using our system. We discuss open problems in Section 6, discuss related work in Section 7, and conclude in Section 8.

2. APPLICATIONS TO WEB SECURITY

Character-level taint tracking can be used to defend against a large class of security vulnerabilities in server-side web code, especially command injection attacks [8, 9, 12, 10].

Many data formats on the web mix both data and control, sending control instructions over the same communication channel as data. This use of in-band signalling introduces the risk that maliciously chosen data might be interpreted by the receiver as control instructions. This is sometimes known as a *command injection* attack. The risk of command injection attack is especially pronounced for text-based message formats, such as those found on the web, as they often provide many subtle ways (e.g., escaping and quoting rules) to induce such confusion.

On the web, common types of command injection attacks include SQL injection, cross-site scripting, HTTP response splitting, path traversal, and shell command injection. For instance, SQL queries mix control (SQL keywords and operators) with data (string and numeric literals), and HTML documents mix control (HTML markup: tags and attributes) with data (text).

Character-level taint tracking is well-suited for detecting injection attacks. Assume that all inputs controlled by the attacker are marked as tainted and taint propagates to every character or substring that depends upon these attacker-controlled inputs. One could then check that the control part of every message is wholly untainted, ensuring that servers cannot be fooled into executing control instructions provided by the attacker. We elaborate on this approach below by explaining how it applies to several classes of command injection vulnerabilities.

2.1 SQL injection

Consider a web application that builds up a SQL query, as follows:

```
query = "SELECT * FROM students WHERE name = '"
        + studentName + "'";
```

Suppose that `studentName` is derived from an input field provided by the user of the web application. If the user enters the name `Bobby`, then the resulting SQL query will select all database entries where the name is Bobby—presumably what the programmer intended. However, if a malicious user claims their name is

```
Bobby'; DROP TABLE students; --
```

then the resulting query becomes

```
SELECT * FROM students WHERE name = 'Bobby'; DROP
  TABLE students; --'
```

This SQL statement deletes part of the database, which is presumably not what the programmer intended.

Character-level taint tracking can help detect and block SQL injection attacks. Before executing any SQL query, the infrastructure should tokenize the SQL query and then check that nothing other than string literals or numeric literals are tainted. In the above example, if the user's name is `Bobby`, then the resulting SQL query will be

```
SELECT * FROM students WHERE name = 'Bobby'
```

where all tainted characters are underlined. This query will be allowed, since tainted characters occur only inside a string literal. On the other hand, when the malicious user enters their name, the SQL query will be:

```
SELECT * FROM students WHERE name = 'Bobby'; DROP
  TABLE students; --'
```

In this case, the infrastructure can recognize that the web application is under attack, since taint is not limited to literals: several SQL keywords (e.g., DROP) and operators (e.g., ;) are partially or wholly tainted. The infrastructure could check every SQL query in this way and decline to execute any SQL query that indicates an attack.

2.2 Cross-site scripting

Many web applications dynamically construct HTML pages through simple string concatenation. This can introduce cross-site scripting vulnerabilities. Consider the following example:

```
html = "<P>Welcome, " + name + "!<BR>";
```

A malicious user who claims their name is

```
Bobby<SCRIPT>alert(42);</SCRIPT>
```

will trick the application into emitting the following HTML:

```
<P>Welcome, Bobby<SCRIPT>alert(42);</SCRIPT>!<BR>
```

In this way, a malicious user could insert malicious Javascript into the page, causing it to be executed with the privileges of the web application.

Character-level taint tracking can help detect and block cross-site scripting attacks. For many web sites, it might suffice to enforce a simple policy on all dynamically constructed HTML: tainted characters may only occur inside text and must not appear in markup, attributes, Javascript, CSS, or other content. A natural relaxation is to allow dynamically constructed HTML to include URLs that are partially or wholly tainted, as long as the protocol is found in a whitelist (e.g., http:, https:, mailto:), and to include other attribute values in certain cases. Many other variations are possible.

2.3 HTTP response splitting

The response to a HTTP request consists of two parts: the HTTP headers and the body (which typically contains the HTML for the page). These may be dynamically constructed by the web application. In particular, headers may incorporate content that was initially supplied by an untrusted user. If an attacker supplies a value that includes a newline character, and if the web application does not sanitize this value before incorporating it into a HTTP header, the attacker may be able to forge a subsequent HTTP header or forge part of the body. In effect, HTTP response splitting is a close cousin to cross-site scripting, except that it occurs in the header part of the response, rather than the body.

Character-level taint tracking can be used to detect and block HTTP response splitting attacks. One approach would be to parse the headers in the HTTP response, checking that no newline (CR or LF) character is tainted and that all header names are wholly untainted.

2.4 Path traversal

Sometimes, web applications access the filesystem through filenames that incorporate some user-supplied information. For instance, consider an application that allows Jimmy to access the files he stores under his account:

```
filename = "/srv/www/users/jimmy/" + filename;
```

Suppose Jimmy is a malicious user and provides the filename `../bobby/secretfile.txt`. Jimmy now has access to Bobby's secret file and potentially access to any file he wants. Path traversal attacks can be detected and blocked using character-level taint tracking. For instance, we can parse each filename accessed by the web application and check whether it contains a tainted slash character.

2.5 Shell command injection

Occasionally, web applications may spawn a new process by passing a command to be executed to the Unix shell. If the shell command incorporates any information supplied by the user, then a malicious user might be able to incorporate shell metacharacters (e.g., `;` or `` ` ``) and trigger execution of malicious commands.

These attacks could be detected and stopped by parsing each shell command and checking that tainted characters appear only in certain positions (e.g., program arguments), and that no metacharacter, operator, or keyword interpreted by the Unix shell is wholly or partially tainted.

2.6 Discussion

Command injection attacks account for a large fraction of vulnerabilities on the web. One study reports that SQL injection, cross-site scripting, and HTTP response splitting account for 52% of observed vulnerabilities in a large collection of web sites [11]; another found that these vulnerability classes accounted for 80% of observed vulnerabilities [4]. Therefore, a successful defense against these attacks could make a significant difference to web applications.

One attraction of the approach to defending web applications sketched above is that it can be applied to legacy applications. The required checks are application-independent and could be incorporated into the underlying application server or libraries. Because this approach detects attacks at runtime as they occur and blocks them before they can cause any harm, no extra effort from developers is needed to deploy these defenses.

This approach crucially relies upon the ability to track taint status at character granularity. Many researchers have proposed taint-tracking mechanisms where there is conceptually a single taint bit per string: each string is either tainted or untainted. Although this may be sufficient for PHP, it is not for Java. In PHP, the programmer may use the `echo` command to avoid direct concatenation, potentially keeping tainted data separate from untainted data. In this case, if taint is checked per `echo` call, string granularity tainting may be sufficient. Java programs, however, generally use string concatenation heavily, so string-level tainting would lead to many false positives. As the examples above illustrate, many SQL queries, HTML documents, and other messages will commonly contain some tainted data. This is often benign and not an indication of an attack. Distinguishing these benign cases from actual attacks requires a way to keep track of which portions of each string are tainted. This motivates our study of character-level taint tracking.

3. TECHNIQUES FOR CHARACTER-LEVEL TAINT TRACKING

Taint tracking can be divided into three parts: source tainting, taint propagation, and sink checking. The first two can be implemented independently of the type of vulnerability considered, but the third depends upon the vulnerability class one wants to defend against. Our prototype currently implements source tainting and taint propagation and omits sink checking.

For every string in a Java program, we track the taint status of each individual character in the string. In order to do this dynamically (and without access to web application source code), we look to what the web application bytecode relies on: Java standard library and the servlet container infrastructure. By altering these modules, our system requires only a one-time change to the server and no subsequent changes when legacy or new web applications are added. Also, because command injection attacks typically arise due to errors in string processing, we focus only on tracking the taint status of strings.

3.1 Source Tainting

We treat all of the information in the HTTP request as untrusted, and mark it as tainted. This includes the HTTP verb (e.g., `GET` or `POST`); the protocol (e.g., `http` or `https`), hostname, and path from the URL requested; form parameters (whether from the URL or from the request body, in the case of POST requests); HTTP headers (including cookies and session information); and the request body. Thus, the HTTP request is the source of all taint.

We implement this policy by instrumenting the web application server, which is responsible for implementing the Java Servlet API. We augment the Servlet API classes to mark all strings from a HTTP request as fully tainted, i.e., we taint every character of those strings. Most of the accessor methods for the request components are contained in the `javax.servlet.http.HttpServletRequest` class. We modify these methods to mark the strings they return as tainted. A few of these accessor methods, however, return a non-String object type, such as a Cookie, Session, or CoyoteReader (a subclass of BufferedReader that lets the servlet read the body of the HTTP request). Therefore, we also modified the `javax.servlet.http.Cookie`, `javax.servlet.http.HttpSession`, and `org.apache.catalina.connector.CoyoteReader` classes to mark these strings as tainted.

3.2 Sink Checking

A sink is a consumer of string data that is security-critical and might be vulnerable to attack if the string is maliciously chosen. We instrument each sink to add runtime checks that use the taint information to determine whether the system is under attack. As outlined in Section 2, these checks are dependent upon the format of the data and the specific type of command injection vulnerability under consideration.

These checks can be added by instrumenting the class libraries. For example, to guard against SQL injection attacks, one can augment the JDBC classes to parse the SQL string and check whether taint is confined to string and numeric literals before the SQL query is sent to the database. For XSS attacks, one can augment the HTTP response class of the web application server to check the HTML response. Our prototype does not yet implement any of these checks. Many researchers have shown that these checks can be implemented efficiently [3, 9, 6, 10], and it would be straightforward to implement them in our framework.

3.3 Taint Propagation

One of the most interesting challenges is to determine how taint should be propagated from string to string, as the application performs various string processing operations. We next describe some of the technical challenges, the taint propagation policy we propose, and the techniques we used to implement this policy.

3.3.1 Goals and Challenges

We had several goals in mind when designing our taint propagation policy:

- *Accuracy:* We want to ensure that the taint propagation policy captures all dependencies between strings.

- *Completeness:* Our taint propagation policy should accurately model all primitive operations that can be performed on strings. Because we cannot predict what types of string processing a legacy web application may perform, we cannot omit any operation from consideration; otherwise we might miss attacks.

- *Backwards compatibility:* We want to ensure that our tainting will not disturb the behavior of the web application in any way. The web application must continue to work properly when we enable taint tracking. In particular, the behavior of the web application with taint tracking enabled should be identical to its behavior without taint tracking, with only one exception: if a sink check detects an attack, then processing may be halted. In other words, the taint mechanism should be transparent: it must not disturb the execution of legacy code. This fail-stop model ensures that we will not "break" legacy web applications.

There are a number of technical challenges in applying these principles to taint tracking of Java applications:

- *Equality:* We must determine the semantics of string comparison: when are two strings identical? If two String objects contain the same sequence of characters, but with different taint status, should they compare equal? For instance, is `"DROP TABLE"` equal to `"DROP TABLE"`? Should the string hashcode depend upon taint information?

- *Mutability:* A closely related question is whether the taint status of a String object is mutable: Can it be changed during the execution of the program? For instance, if we have a String object representing the string `"DROP TABLE"`, should we provide a way to set the taint bits for individual characters?

- *Interning:* Java distinguishes between object identity (which may be compared with the `==` operator) and equality (which may be compared with the `equals()` method). In particular, it is possible to have multiple String objects that are not identical (`s != t`) but that have the same contents and thereby compare equal (`s.equals(t)`). As a performance enhancement, Java provides a way to intern strings: given a String object `s`, `s.intern()` returns a canonical String object with the same contents as `s`, with the promise that if `s.equals(t)`, then we will have `s.intern() == t.intern()`. This is sometimes used to speed up string comparisons. Should taint status affect interning? If we have two String objects that represent the same sequence of characters, but with different taint status, should interning them yield the same canonical String object? How should the taint status of `s.intern()` relate to the taint status of `s`?

- *Reflection:* Web applications may use reflection to invoke methods at runtime, rather than calling them directly. We would like those web applications to continue to work correctly when taint tracking is enabled.

- *Serialization:* If the web application serializes a String object, what should happen to the associated taint information? Should the serialized representation of strings be modified to store their taint status?

- *Character encodings:* Java strings are specified to be a sequence of Unicode characters, represented in UTF-16. UTF-16 is a variable-length representation: some characters are represented in two bytes, and others take four bytes. Variable-length representations have been known to introduce vulnerabilities in other contexts, such as UTF-8 (another variable-length representation). For instance, UTF-8's multibyte encodings provide one potential way to bypass quoting. Suppose a web application tries to prevent SQL injection attacks by replacing each single quote (`'`) with an escaped version (`\'`). Consider the input string `John O♭'Neill`, where ♭ denotes the byte `0xC0`. After escaping, the input becomes `John O♭\'Neill`. In

UTF-8, `0xC0` indicates the start of a multibyte sequence, so if the developer is not careful, the database might treat ♭\ as a two-byte sequence encoding a single Unicode character, leaving the single-quote unescaped. We would like our taint propagation policy to assist in defending against such attacks.

- *Locales and internationalization:* In Java, the interpretation of strings depends upon the locale. In certain non-English locales (particularly Turkish, Azeri, and Lithuanian), several string operations might convert a four-byte character character to a two-byte character or vice versa. We must ensure that this does not provide a subtle way to evade taint tracking or to launder tainted data into untainted data.

3.3.2 Taint Propagation Policy

Basic policy. Conceptually, for each string, we associate a separate taint bit with each character of the string, indicating whether that character was derived from untrusted input or not. We instrument the string-related library classes (String, StringBuffer, and StringBuilder) to record this taint information, and we modify their methods to propagate taint information from string to string. In some cases, we added additional constructors to enable creation of strings with a specified taint status. We note that this approach is compatible with web applications that use reflection to dynamically invoke methods: those applications will continue to work without change.

For most of the methods (e.g., replace(), concat(), substring(), trim(), toUpperCase(), etc.), propagating taint information is straightforward. We discuss non-trivial cases below.

Equality and interning. To ensure backwards compatibility and avoid breaking legacy web applications, we decided that the conditions under which two String objects compare equal should be identical, regardless of whether taint tracking is enabled or not. As a consequence, our instrumented `equals()` method does not examine the taint information: two String objects compare equal if they represent the same sequence of characters, even if their taint status differs. Similarly, the `hashCode()` method ignores taint information when computing taint status. As a result, neither the `equals()` nor `hashCode()` methods are modified in our prototype.

Similarly, we decided that interning should ignore the taint information. This ensures that taint tracking will not disturb the behavior of the web application, and thus avoids breaking legacy web applications. This does come at a potential cost in accuracy: it means that taint information does not necessarily propagate from the string `s` to the result `s.intern()`. This could cause taint to be lost if the program first adds an untainted string to the intern pool, then later calls `intern()` on another String object with the same contents. For instance:

```
String s = "foo"; // untainted
String t = s.intern();
String u = makeTaintedCopy("foo"); // tainted
String v = u.intern();
// Now t == v, so v is untainted
```

If the web application uses interning, this could cause false negatives: if we are unlucky, in principle attacks on the web applications might go undetected. However a mitigating factor is that this may be very difficult to exploit: it seems unlikely that there will be an untainted string already present in the intern pool that happens to be just what an attacker needed to mount a SQL injection or cross-site scripting attack.

Our treatment of interning could also cause false positives. For similar reasons, we expect this to be rare.

Mutability. Since String objects are immutable in Java, we decided that their taint status should be immutable as well. Instead of providing a way to set or clear the taint bits on a string, we instead provide a way to create a new String object with the same sequence of characters as the original but with a different taint status. We found that this was sufficient to encode all of the patterns we encountered.

Primitive characters. In most cases, when propagating taint information, we directly copy the taint status of the source character to the destination character. We were forced to make two exceptions for the following methods, from `java.lang.String` and `java.lang.StringBuilder`, respectively:

```
public String replace(char oldChar, char newChar);
public void setCharAt(int index, char ch);
```

One of the difficulties of taint tracking through the standard library classes is that it is not possible to track the taint status of primitive types, such as `char`, `byte`, or `int`. Consequently, we lack taint information on single `char`s. To avoid false positives, we decided to treat `char`s and other primitive types as untainted. This may introduce false negatives, i.e., missed attacks, as discussed in Section 3.4.

This introduces challenges for modelling the `replace()` and `setCharAt()` methods mentioned above. Because we treat primitive `char`s as untainted, in our basic approach these methods would replace a character (tainted or not) from the underlying string with an untainted char (`newChar` or `ch`). In other words, our basic approach would clear the taint bit on the character of the string that was replaced, resulting in loss of taint information and, potentially, missed attacks. To avoid missing attacks, we decided to refine our standard approach for these two methods. Instead of clearing the taint bit on the character in the string that is replaced, we preserve the taint status of the original character. Thus, these two methods leave the taint status of the string being operated upon unchanged.

Serialization. For backwards compatibility, we chose not to serialize taint information. In particular, we wanted to ensure that a web application with taint tracking enabled would still be able to read data it might have serialized earlier before taint tracking was enabled and vice versa. This could be a source of taint information loss, but we hypothesize that due to the performance requirements of interactive websites, very few web applications, if any, serialize request information. Consequently, we did not modify any of the serialization or deserialization methods (e.g., `readObject()` or `writeObject()`).

Character encodings. The Unicode standard defines a universal character set. Every Unicode character is assigned a code point, a value ranging from 0x000000 to 0x10FFFF. To take advantage of the fact that most characters can be represented with fewer than 4 bytes, a few variable-length encoding schemes have been proposed. Java, in particular, uses UTF-16, where some code points are represented in two bytes and some in four bytes. In Java, the primitive `char` type holds a single code unit: a two-byte value. Each Unicode code point is represented in UTF-16 as either one or two code units (`chars`). The set of characters that can be represented by a single Java code unit is called the Basic Multilingual Plane. Characters that must be represented by two Java code units are called supplementary characters. A supplementary character is represented as two code units, where the first code unit takes on a value from 0xD800 to 0xDBFF and the second Java code unit ranges from 0xDC00 to 0xDFFF. The range from 0xD800 to 0xDFFF is reserved for UTF-16's supplementary characters and does not represent any real character. The set of code points that can be represented in a single code unit (the Basic Multilingual Plane) are exactly those in the range U+0000 to U+D7FF or U+E000 to U+FFFF.

Conceptually, a String stores Unicode characters (code points). Internally, this is represented as a `char` array holding a sequence of code units representing the UTF-16 encoding of the Unicode characters of the string. With this scheme, one can examine a single code unit in a String and tell whether it is a single unit, the first part of a two unit code, or the second part of a two unit code: one does not need to examine any other context (e.g., the preceding or following code units).

Conceptually, when processing Strings, we consider each character wholly tainted or wholly untainted. Our internal representation stores a taint bit with each code unit. However, if a string contains a supplementary character, i.e., a character that is represented as a pair of code units, then we consider the character tainted if either of its two units are marked tainted. This, combined with the fact that non-supplementary characters are represented with a two-byte value that is disjoint from the values used to encode supplementary characters, helps defend against attacks that attempt to bypass quoting.

Locales. Some locales (particularly Turkish, Azeri, or Lithuanian) support accents or diacritical marks using combining characters. These allow a two-code unit sequence to represent a single character: one code unit represents the letter, and the other code unit represents the accent, diacritical mark, or other combining character. However many of these characters can also be represented in a single code unit. For various reasons, it can be useful to canonicalize the representation. As a consequence, some Java string methods (such as String.toUpperCase() and toLowerCase()) may convert a two-code unit character to a one-code unit character or vice versa in certain cases. In this case, we take the logical OR of all the code units' taint information to determine the taint status of the new character. In other words, if part of a two-code point character is tainted, the combination is itself considered tainted.

3.3.3 Taint Propagation Mechanism

Representing taint information. The String data type in Java is an immutable data type that consists of a character array, an offset into the array (starting index), count (length of the string), and hashcode:

```
public final class String {
  private final char[] value;
  private final int offset;
  private final int count;
  private int hashCode;
  ...
}
```

A String object `s` represents the string consisting of the sequence of characters in the range `s.value[s.offset .. s.offset+s.count-1]`. This representation permits multiple Strings to share a single character array while referring to different (sub)strings by using different offset and count values.

We augment the String class by adding a boolean array to store the taint value for each character of the string:

```
public final class String {
  private final char[] value;
  private final int offset;
  private final int count;
  private int hashCode;
  private boolean[] taintarr;
  ...
}
```

We consider `s.value[i]` tainted if and only if `s.taintarr[i]` is true. This increases the space required to represent a String by a small constant factor.

We modify the methods of the `String`, `StringBuffer`, `StringBuilder` classes as described above. We found that only a subset of methods handled taint information and thus needed to be instrumented; we examined every method and instrumented each one that needed to propagate taint information. Of 96 original methods in the `java.lang.String` class, we instrumented 28 methods and added 11 methods. Of 60 original methods in the `java.lang.StringBuilder` class, we instrumented 27 methods and added 6 methods. Of 60 original methods in the `java.lang.StringBuffer` class, we instrumented 26 methods and added 4 methods.

We then replace the old string class files contained in the Java standard library JAR file with the new, augmented class files. When a Java program is executed, the JVM automatically loads the augmented classes instead of the original string classes. As the propagation policy is implemented in the library classes, it does not require any changes to the legacy Java application's source code or bytecode. This makes the solution easy to adopt.

Optimizing for the common case. We noticed that many strings are wholly untainted: none of their characters are marked as tainted. To optimize our taint tracking mechanism for this common case, we perform character-level taint tracking on demand only. Instead of introducing a taint array for every string, we introduce character taint information only if one or more of the characters of the string are marked as tainted. We set the `taintarr` field of wholly untainted strings to null. This eliminates unnecessary taint propagation for string literals and other strings from trusted sources, and avoids unnecessary construction and copying of taint arrays for wholly untainted strings, thus reducing the runtime overhead.

This is an advantage of our method over other methods that use shadow memory for every character [12]. This optimization is made possible because Java strings are represented by a dedicated String class, which serves as an abstract data type that we can easily instrument. It might be more difficult to apply this kind of optimization in a language like C.

3.4 Discussion

With our technique, it is possible to lose taint information in a few subtle ways, which can lead to a failure to detect some attacks.

We do not track taint information associated with the Java `char`, `char[]`, or other primitive types. If the web application developer takes a tainted string from the HTTP request and extracts its character information (through, e.g., `getBytes()`, `getChars()`, `toCharArray()`, `codePointAt()`, `charAt()`, etc.) and then performs text processing on these primitive `char`s, then taint information is lost. We hypothesize that this is a rare occurrence if the developer is not adversarial. One way to evaluate this hypothesis might be to log when a tainted character is extracted from a string using one of these methods, to measure how frequently this scenario occurs in practice. Our prototype currently logs these cases, but we have not yet performed any experiments to test this hypothesis.

There is one case where taint information can be lost due to an inability to taint a character buffer from the Servlet framework. In the implementation of the `javax.servlet.http.HttpServletRequest` interface, the getReader() method returns an object of type CoyoteReader which allows for access to the raw request body data in the form of either a String or character array. If returned as a String, taint can be tracked. However, if it is accessed as a character array, taint information is lost. We hypothesize that the latter case would usually occur only for opaque handling of binary data (e.g., file uploads) that is unlikely to participate in command injection attacks. We do not expect applications to use getReader() to obtain form parameters, as that data is more naturally and easily accessed via the getParameter() method.

Another way to lose taint information is if we fail to propagate taint or miss a taint source. Outside of methods in the StringBuffer and StringBuilder classes, a few String methods create objects from other classes. The String.format() method creates an object of type `java.util.Formatter`. This method constructs a string from arguments given in the printf() style. For this, the format() in the formatter class also needs to be augmented to track taint across any %s style strings. Likewise, a few methods call other methods in the `java.util.regex.Pattern` and `java.util.regex.Matcher` classes. We have yet to instrument these additional classes.

As mentioned earlier, it is also possible to lose taint information through serialization. Finally, like other approaches, we do not attempt to track implicit data flow.

4. IMPLEMENTATION

Our taint tracking system was implemented using the 32-bit IBM JDK 6.[1] We experimented with other JDKs, including Sun's OpenJDK 6 and IBM JDK 5. However, both of these JDKs presented problems when trying to augment the string classes. We suspect that they require a predetermined size for the string class files, so changes to these classes broke the JVM.

To support web applications, we instrumented Tomcat Servlet Container 6, which follows the Servlet 2.5 specification.[2]

To ensure that our implementation correctly implements the taint propagation policy described in Section 3.3, we wrote over 135 unit tests and used them to validate our implementation.

5. EXPERIMENTAL RESULTS

Our web application server was run on a 2.40GHz Intel Pentium 4 / 1 GB RAM machine. The clients were run on 4 Dual-Core AMD Opteron Processor 2214 machines.

To test our system, we used the JForum web application, a Java-based discussion board. We believe this to be an appropriate web application for our preliminary experiments as it allows us to create a range of request types: basic GET requests, basic POSTs of varying length, and GETs and POSTs that require varying amounts of additional text processing (due to handling URL links, emoticons, images, etc.). We examine and compare the request throughput of the program under various conditions. We limit our focus to overhead created by source tainting and taint propagation.

For measurement purposes, we examined the roundtrip time of making the request, processing the request on the server, and receiving the response. With less than 100 KB per response (and much smaller requests), 100+ Mbps connections, and measured response times on the order of several milliseconds, the network effect can be considered negligible compared to the server-side computation.

In Figure 1 we show the throughput as the size of the tainted data in the POST is increased in both the instrumented and the uninstrumented versions of Tomcat. As predicted, the overall throughput decreases as the POST increases. More importantly, we also see that there is very little difference between the throughput for JForum with and without taint tracking.

In Figure 2, we examined the throughput for 1) the uninstrumented program, 2) the instrumented program with the on-demand character-level taint tracking optimization, and 3) the naïve character-level taint tracking for different types of requests: basic GET requests, GET requests with additional text processing, basic POSTs, and POSTs with additional text processing. These experiments were run five times with 1000 requests each, averaging the latter four hot-cache runs. As expected, the optimized taint tracking method performs slightly better than naïve taint tracking, and the uninstrumented version performs just slightly better than the optimized taint tracking. In fact, in most cases the runtime overhead of the optimized version is less than 5% (see Table 1).

[1] One thing to note about the IBM JDK is that it structures its library class files differently from those of the Sun JDK. While Sun stores all of its class files in a JAR file called rt.jar, IBM breaks the library class files down into separate JAR files. The relevant string library class files are located in vm.jar.

[2] Due to a bug in Tomcat 6, it can only be built with Java 1.5. Because the instrumentation could not be done with Java 1.5, we partially built under 1.5 (Part 1 "ant download"), and then completed the build with 1.6 after that point (Part 2 "ant"). Feel free to contact the authors for more information about this workaround.

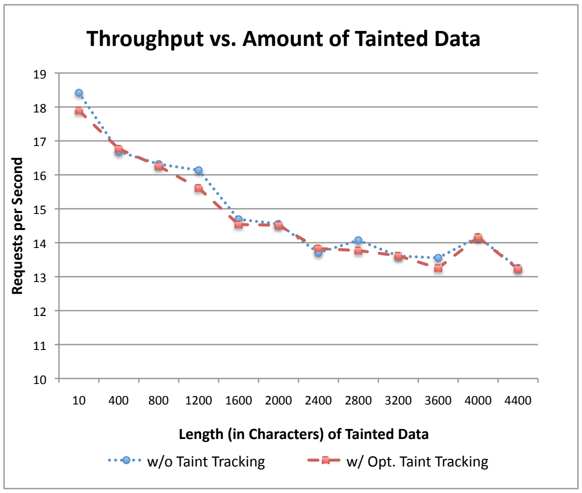

Figure 1: Even as the length of the JForum text increases, there is very little difference between the throughput for JForum with and without taint tracking.

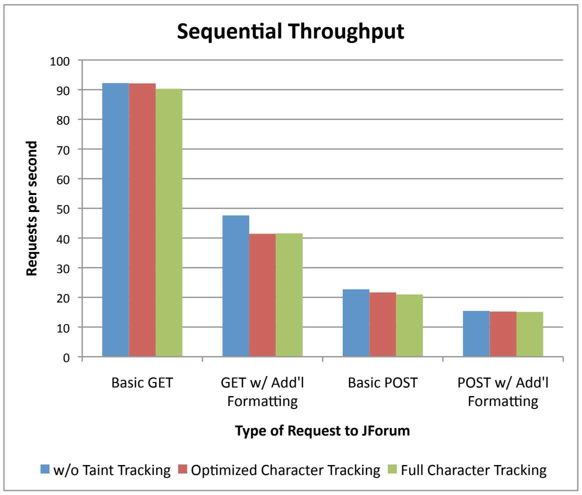

Figure 2: Throughput for different types of requests to JForum without taint tracking, with optimized taint tracking, and with unoptimized taint tracking

Table 1: Measured Overhead

Request Type	w/ Opt	w/o Opt
Basic GET	0.11%	2.11%
GET w/ Add'l Formatting	14.91%	14.54%
Basic POST	4.81%	8.28%
POST w/ Add'l Formatting	1.45%	2.20%

There are a few limitations of our analysis. In our overhead experiments, the HTTP requests were sent sequentially and therefore may not have driven the server to its maximum serving capacity. Additionally, our workload was artificially created, and therefore it does not test the system under a real mix of HTTP request types and lengths. By individually examining GETs and POSTs separately, we attempt to determine the runtime overhead of the program at the two ends of the spectrum.

Due to our decision to leave the choice of the sink checking technique to the user, we could not perform an analysis of the rate of false positives and negatives. However, we expect to see similar results to results seen in other character-level taint tracking implementations. The work of Halfond et al. [6] as well as Sekar [10] both achieve a false positive rate of 0%.

Our preliminary experiments indicate that character-level taint tracking can be performed efficiently with a runtime overhead ranging from 0–15%. While this is only one application tested on artificial workloads, we show that our approach is a promising step towards efficient taint tracking.

6. OPEN PROBLEMS

There are many open problems with regard to potential applications to taint tracking in web services. Many programs are starting to integrate multiple programming languages. An interesting area to examine is taint tracking across different languages. Another open problem is considering how to track taint across a database. The current focus is to perform taint checking before input enters the database. Instead, to check for persistent XSS attacks, we can consider tracking taint across the database. Finally, many systems examine XSS attacks from the server-side. The problem with this is that different browsers might interpret the HTML differently. One solution would be to specify taint status to the browser through the HTML. Conversely, another approach may be to examine XSS attacks from the server-side using the tainted data to predict where potential attacks may exist.

7. RELATED WORK

Our work is inspired by seminal work on fine-grained taint analysis in other languages, including PHP, C, and Java.

PHP. Nguyen-Tuong et al. show how to provide character-level taint tracking for PHP web applications by augmenting the PHP interpreter [8]. Their implementation achieves a performance overhead of less than 10%. Pietraszek and Berghe present a similar character-level taint tracker for PHP, and use it to prevent SQL injection [9]. The runtime overhead of their scheme is 2–17%, depending upon the application. Futoransky et al. present GRASP, an augmented PHP interpreter that also does character-level taint tracking and achieves a 30–100% overhead [3].

C. Several researchers have studied byte-level taint tracking of C programs. Xu et al. use source-to-source transformations to track the taint status of every byte of memory [12]. By applying their transformation to the source code of the PHP interpreter, they also show how to protect PHP web applications from a number of vulnerabilities. They do not measure the performance overhead in this scenario, but overheads for other applications range from 3–100%. Lam and Chiueh present a general framework which can be used for byte-level taint tracking, among other things [7]; they observe performance overheads in the 1–30% range. Chang et al. further reduce performance overheads by applying static analysis to optimize the dynamic taint tracking operations [1]. They achieve performance overheads as low as 0–4% for taint tracking of several network servers written in C.

While the performance of these systems is impressive, their performance cannot be directly applied to our measurements because they evaluated different systems and different workloads. Also, few web applications are written directly in C, which limits the ability to apply C-based systems to web security directly. It is plausible that the static analysis methods developed by these authors could be applied to optimize character-level taint tracking in Java as well, though we have not investigated this direction.

Java. Although we have seen impressive techniques for taint tracking in other programming languages, it does not necessarily follow that they can or will translate well to other languages. As we have discussed previously in Section 2.6, PHP may be more suited to string-level taint tracking because of its ability to keep tainted data separate from untainted data by using the `echo` command on separate strings. Java however uses string concatenation heavily and therefore could be prone to false positives if taint is not tracked at the character granularity. With regard to C, the language lacks string types, possibly requiring a more heavyweight process to perform taint tracking.

A few researchers have examined taint tracking for Java. The early work only tracked taint at a string level and was not able to track taint information at a character granularity [2, 5].

Since then, Halfond et al. have introduced methods for character-level taint tracking of Java applications based upon bytecode rewriting [6]. They apply their methods to preventing SQL injection in web applications, and observe a performance overhead of 1–19%. Our work shows that one can achieve acceptable performance simply by replacing library classes, without requiring bytecode rewriting of the application itself. Additionally, our work handles taint propagation for Java reflected strings while the bytecode rewriting technique is unable to do this.

8. CONCLUSION

In efforts to harden legacy web applications against input-based attacks, we introduce a taint tracking system that can be easily adopted and deployed through server-side changes. This approach is a one-time modification that is transparent to the web application. It neither requires any web application source code nor any change to the bytecode.

Our preliminary experiments indicate that character-level taint tracking can be performed efficiently without a significant impact on the application's runtime performance: in the context of the JForum application, we observed a runtime overhead of 0–15%. Although this paper focuses on the problem of securing web applications, our approach can also be applied to track taint information in any Java application.

Acknowledgments

We would like to thank Koushik Sen and Prateek Saxena for their guidance and invaluable feedback. We also thank Brian Chess, Matt Finifter, Adrian Mettler, Cynthia Sturton, and the anonymous reviewers for their comments on earlier versions of this paper. This research was supported by NSF grants CNS-0524745 and CNS-0430585.

9. REFERENCES

[1] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 39–50. ACM New York, NY, USA, 2008.

[2] B. Chess and J. West. Dynamic taint propagation: Finding vulnerabilities without attacking. *Information Security Technical Report*, 13(1):33 – 39, 2008.

[3] A. Futoransky, E. Gutesman, and A. Waissbein. A dynamic technique for enhancing the security and privacy of web applications. *Proc. Black Hat USA*, 2007.

[4] J. Grossman. WhiteHat website security statistics report, Aug. 2008. `http://www.whitehatsec.com/home/assets/WPstats0808.pdf`.

[5] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Annual Computer Security Applications Conference (ACSAC 2005)*, pages 303–311, 2005.

[6] W. Halfond, A. Orso, and P. Manolios. WASP: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering (TSE)*, 34(1):65–81, 2008.

[7] L. Lam and T. Chiueh. A general dynamic information flow tracking framework for security applications. In *Annual Computer Security Applications Conference (ACSAC 2006)*, pages 463–472, 2006.

[8] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Informations Security Conference (SEC 2005)*, pages 295–307. Springer, 2005.

[9] T. Pietraszek and C. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID 2005)*, volume 3858 of *Lecture Notes in Computer Science*, page 124. Springer, 2006.

[10] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Network and Distributed Systems Symposium (NDSS 2009)*, Feb. 2009.

[11] Web Application Security Consortium. Web Application Security Statistics Project 2007. `http://www.webappsec.org/projects/statistics/wasc_wass_2007.pdf`.

[12] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, pages 121–136, August 2006.