DBTaint: Cross-Application Information Flow Tracking via Databases *

Benjamin Davis
University of California, Davis

Hao Chen
University of California, Davis

Abstract

Information flow tracking has been an effective approach for identifying malicious input and detecting software vulnerabilities. However, most current schemes can only track data within a single application. This single-application approach means that the program must consider data from other programs as either all tainted or all untainted, inevitably causing false positives or false negatives. These schemes are insufficient for most Web services because these services include multiple applications, such as a Web application and a database application. Although system-wide information flow tracking is available, these approaches are expensive and overkill for tracking data between Web applications and databases because they fail to take advantage of database semantics.

We have designed DBTaint, which provides information flow tracking in databases to enable cross-application information flow tracking. In DBTaint, we extend database datatypes to maintain and propagate taint bits on each value. We integrate Web application and database taint tracking engines by modifying the database interface, providing cross-application information flow tracking transparently to the Web application. We present two prototype implementations for Perl and Java Web services, and evaluate their effectiveness on two real-world Web applications, an enterprise-grade application written in Perl and a robust forum application written in Java. By taking advantage of the semantics of database operations, DBTaint has low overhead: our unoptimized prototype incurs less than 10-15% overhead in our benchmarks.

*This research is partially supported by NSF CNS award 0644450 and by an AFOSR MURI award.

1 Introduction

Information flow tracking has been very successful in protecting software from malicious input. The program identifies the sources of untrusted input, tracks the flow of such input, and prevents this input from being used in security sensitive contexts, such as the return addresses of function calls or the parameters of risky system calls [16, 19]. Currently there are two types of information flow tracking mechanisms: application-wide and system-wide. The former tracks information flow within the same application [16, 19], while the latter tracks information flow in the entire operating system [17].

As computation moves to the Web, Web services have become highly attractive targets to attackers. In fact, attacks involving malicious input to Web applications, such as Cross-site Scripting (XSS) attacks, are among top software vulnerabilities [4]. Information flow tracking is a logical approach for preventing these attacks by tracking malicious input [10, 14]. However, the application and effectiveness of information flow tracking is limited by the only two types of current mechanisms: single-application and system-wide tracking.

A typical Web service consists of multiple applications, such as a Web application, which implements business logic and generates Web pages, and a database, which stores user and application data. In multi-application settings like Web services, single-application information flow tracking is inadequate, as it would force Web applications to decide between treating all the results of database queries as tainted or treating them as untainted. This would inevitably result in false positive or false negative when the database contains both tainted and untainted data.

To enable cross-application information flow tracking, one might resort to system-wide information flow tracking systems. However, there are several problems with these systems. They track more information than needed for protecting Web services, which comes at an unnec-

essary performance cost. Protecting against XSS attacks requires tracking information flow only in the Web application, database, and the information flow between them, rather than in every operation in the entire system. Also, these system-wide mechanisms fail to take advantage of application semantics. Without the high-level semantics, these systems cannot properly perform taint propagation throughout complex database operations.

We introduce DBTaint, a system that provides information flow tracking across databases for enabling cross-application information flow tracking. DBTaint extends the database to associate each piece of data with a taint tag, propagates these tags during database operations, and integrates this system with existing single-application taint tracking systems. By providing this integration via SQL rewriting at the database client-server interface, DBTaint is completely transparent to Web applications. This allows developers to use DBTaint with existing, real-world legacy applications without modifying any Web application code.

DBTaint can provide more accurate information flow tracking than single-application taint tracking systems. Services using DBTaint will have fewer false positives than systems that consider all values from external applications (like databases) as tainted. Similarly, services using DBTaint will have fewer false negatives than systems that consider all values from the database as untainted. Furthermore, since DBTaint tracks taint propagation inside the database, it takes advantage of database semantics to track taint propagation accurately. Besides providing a more accurate taint tracking system, DBTaint can also be used to detect potential vulnerabilities in applications. If user input is untainted during sanitization in a Web application, inspecting the taint values of database columns may reveal subtle security vulnerabilities. Our insight is that if a column in the database contains both tainted and untainted data, it may signal incomplete sanitization in the database client, e.g., when user input is santized only on a subset of program paths. Such observation should alert the programmers to audit the sanitization functions in the program carefully.

We make the following contributions:

- We design and implement a system that tracks information flow across databases. This allows cross-application information flow tracking to protect Web applications from malicious user input.

- We improve on single-application taint tracking systems by reducing false positives/false negatives, and improve on system-wide taint tracking systems by tracking only the taint propagation that matters to the target application and by taking advantage of database semantics to improve tracking accuracy.

- Our system is also useful for analyzing certain behavior of database client applications, such as identifying potential incomplete sanitization in Web applications.

- We design a flexible system to integrate single-application taint tracking systems with the PostgreSQL database. This system allows legacy applications to take advantage of our system transparently.

- We implemented two prototypes of DBTaint that work with real-world Web applications written in Perl and Java. These prototypes, although unoptimized, have low performance overhead.

2 Design

DBTaint is a system that allows developers to track information flow throughout an entire Web service consisting of Web applications and database servers. DBTaint provides information flow tracking in databases and integration with single-application information flow mechanisms. The system is completely transparent to the Web applications, which do not need to be modified in any way to take advantage of DBTaint.

We assume that the developer is benign, and has the ability to replace the database interface and database datatypes used in the Web service with the modified (DBTaint) versions. We also assume that the single-application taint tracking engine(s) used for each individual Web application appropriately taints input from unsafe sources (e.g. user input).

DBTaint propagates the taint information throughout the multi-application system, but does not attempt to actively prevent the program from operating unsafely. Rather, by propagating and maintaining taint values for each piece of data in the system, we provide developers with the information needed to perform the sink checking and handling appropriate for their setting. Although DBTaint was motivated by Web services, our prototypes provide cross-application information flow tracking to any multi-application setting where applications communicate via databases. For brevity, we will refer to these applications as Web applications onward.

2.1 Taint Model

2.1.1 Soundness

DBTaint helps improve the security of Web services by tracking the trustworthiness of all the data values used by the service. DBTaint marks each value as either *untainted* or *tainted*. DBTaint marks a value as *untainted*

only if it can determine that the value is trusted. Therefore, when DBTaint marks a value as *tainted*, it could be because DBTaint has determined that the value is indeed untrusted or because DBTaint cannot determine whether the value is trusted.

We say two values are in the same *context* if they belong to the same column or their respective columns are compared in a `JOIN` operation. With DBTaint, the database marks an output value as untainted only if there was an occasion when the same value in the same context was marked as untainted when it entered the database, or if the output value is derived from untainted values only.

The above property implies that:

- If a context contains two identical values but one is tainted and the other is untainted, DBTaint may return this value either as tainted or untainted. Our prototype chooses to always return this value as untainted to improve the accuracy of taint tracking in the Web application.
- DBTaint will never return a value as untainted if this value has never entered the context as untainted and is not derived from only untainted values.

2.1.2 Scope of Taint

DBTaint can work with any taint tracking mechanism inside the Web application. In the simplest, and most common, case, the taint value of a data when it exits the database is the same as its taint value when it enters the database. Consider an SQL query for the `MAX` of two values in the database, where one value is 3 and tainted, and the other value is 5 and untainted. DBTaint returns the value 5 to the database client (the Web application) untainted, because the value 5 was untainted when it entered the database. Similarly, data in the result of a `JOIN` query carry their taint values in the database, regardless of the taint values of other data (e.g., data in the common columns during `JOIN`) that may have affected the selection of the data in the result.

2.1.3 Backward Compatibility

We adopt the principle of "backwards compatibility", similar to the one described by Chin and Wagner [7], and design DBTaint such that a DBTaint-unaware application should behave exactly the same regardless of whether it is retrofitted with DBTaint. Under this principle, when DBTaint compares two data, it ignores their taint values. Besides ensuring backward compatibility, this principle also allows DBTaint to set the taint value of certain output data more accurately. For example, consider computing the `MAX` of a tainted value 2 and an untainted value 2. Either value is an acceptable result for this query, but we choose to return the untainted value. Similarly, in a `SELECT DISTINCT` query, we again prefer to return untainted versions of equal values when available.

2.2 Information Flow Tracking in the Database Server

Because current mainstream database systems do not natively provide a mechanism for storing taint information associated with each piece of data, DBTaint provides a mechanism for storing this information without losing precision of the original data. Furthermore, DBTaint propagates the taint information for database values during database operations.

2.2.1 Storing Taint Data

DBTaint provides information flow tracking capabilities in databases at the SQL level, requiring no changes to the underlying database server implementation. Capabilities added to the database at the SQL-level are likely simpler and more portable than those made by modifying the underlying implementation of a particular database server. Furthermore, by utilizing SQL to maintain and operate on the taint information, we avoid the need to provide new mechanisms to insert, retrieve and operate on taint information in the database server.

Many databases support *composite* data types, where each data cell may store a tuple of data. We used this feature to store taint information alongside associated data values, allowing DBTaint to use the well-understood SQL API for interacting with these taint values. The additional functionality (like auxiliary functions) DBTaint needs to add to the database can be done at the SQL-level as well (e.g. via `CREATE FUNCTION`).

Compared to alternative implementation approaches (e.g. storing taint bits in mirrored tables or additional columns), we hypothesized that composite types would be the simplest. It allowed our SQL-rewriting operations to rewrite each original query into exactly one new query, avoiding the need for extra queries to maintain mirrored tables. Also, using composite types allowed us to build taint propagation logic into the database type system rather than into each rewritten query.

2.2.2 Operating on Taint Data

In addition to creating the database composite types, DBTaint provides some database functions that make operating on these types simpler. We provide the database functions `getval()` and `gettaint()` to extract the data and taint values from a DBTaint tuple, respectively. These functions are used in the SQL rewriting phase, described in Section 2.4.1. DBTaint also provides neces-

sary database functions for these composite types (e.g. equality and comparison operators). Finally, DBTaint provides functions to propagate taint values in aggregate functions (like `MIN` and `MAX`) and arithmetic operations (when one or more operand is tainted, the result is tainted).

2.3 Information Flow Tracking in the Database Client

DBTaint leverages existing single-application information flow tracking systems to manage taint information in the client, and integrates the single-application taint tracking system with the new database server functionality at the interface between the two applications. DBTaint works with any mechanism for taint tracking in the database client (the Web application). For instance, we have implemented a version of DBTaint for Perl that uses a modified version of Perl's taint mode. We also developed a prototype version of DBTaint that uses an efficient character-level taint tracking system for Java [7]. While the single-application taint engines propagate taint throughout the single application, DBTaint handles the propagation of this taint information across application boundaries when this data is used in a SQL query.

Many other single-application taint tracking systems exist, and DBTaint can be easily extended to integrate with these engines as well. For example, there also exist preliminary implementations of PHP with support for tainted variables.[1]

2.4 Database Client-Server Integration

Once we can track the information flow within a single application and within a database, DBTaint must provide a way to propagate the taint information between database client applications and the augmented database server. While we could perform this by modifying the Web application directly, this approach does not scale well, as the user would need to modify every new Web application individually. Furthermore, the amount of work required to make the changes would scale with the size and complexity of each Web application.

Instead, DBTaint integrates the information flow systems of the database client and database server at the interface between these two systems. For example, Perl programs generally use the DBI (DataBase Interface) module to access database servers, and Java applications often use JDBC (Java DataBase Connectivity) API. By adding our DBTaint functionality at these interfaces, we can integrate the taint tracking systems of multiple applications completely transparently to the Web application.

DBTaint requires three changes to the database interface:

- Rewrite all queries to add additional placeholders for taint values associated with the data values, and to add appropriate taint values where appropriate.
- When the application supplies the parameter values, determine and pass the corresponding taint values.
- When retrieving the composite tuples from the database, collapse them into appropriately tainted data values then return them to the Web application.

2.4.1 Rewriting SQL Queries

In DBTaint, the database server tables are composed of composite values that contain both the data and the taint value associated with that piece of data. However, since the Web applications that use these databases are not modified in any way, their data values and corresponding SQL queries do not include the necessary information to maintain the data taint values in the database. A key component of the DBTaint system is the way the SQL queries from the Web application are dynamically rewritten to propagate taint information between the Web application and the database server transparently to the database client.

DBTaint performs two main types of transformations on portions of SQL queries: *tupling* and *flattening*. These operations performed on the appropriate parts of a SQL query before passing it through to the database server.

Tupling is the process of taking a data value and converting it into a tuple that contains the original value and the associated taint value. For example, when a Web application sends an `INSERT` query that includes a data value to the database interface, DBTaint rewrites that portion of the query into a tuple containing the data value and the taint value of that data. If the Web application passes a parameterized query (with `?` placeholders for data values to be supplied later), DBTaint rewrites the query to include additional placeholders for the corresponding taint values.

Assume we specify a composite type using the PostgreSQL syntax: `ROW(x,y)` where `x` is the data value, and `y` is the corresponding taint value. If the Web application passes the following query to the database:

```
INSERT INTO posts (id, msg) VALUES
(1, ?)
```

then DBTaint rewrites this query to include the taint value of the `1` substring (e.g. 0 if untainted), and adds a place for the taint value of the message data to be supplied later.

```
INSERT INTO posts (id, msg) VALUES
(ROW(1,0), ROW(?,?))
```

Flattening is the process of taking a tuple value in the database and removing the associated taint value when it is unneeded. We have designed DBTaint such that using the system does not change the behavior of the Web application. So, sometimes it is necessary for DBTaint to extract only the data value for certain SQL operations in order to perform the appropriate operations. For example, if the Web application wishes to select rows where a specific column is equal to a hardcoded value, then we disregard the taint value during the selection process.

For example, when the Web application issues the request:

```
SELECT username FROM users WHERE
user_id = 0
```

Since in this case the taint value of the `user_id` field is unimportant, DBTaint extracts only the data value and the query becomes:

```
SELECT username FROM users WHERE
getval(user_id) = 0
```

2.4.2 Rebinding Parameterized Query Values

Applications often use parameterized queries for defense against SQL injection attacks, improved performance, and increased maintainability. Parameterized queries use placeholders for parameters that the Web application passes later. Often DBTaint must augment these queries by adding additional placeholders for the corresponding taint values. Unfortunately, this means that the index-based bindings the Web application uses may no longer be valid (e.g. binding a value to placeholder three may no longer be the third parameter in the rewritten query). Furthermore, because the Web application does not know about these new taint parameters, DBTaint must provide them to the database.

When a Web application attempts to bind a parameter to a particular position in a SQL query, DBTaint intercepts this request and computes the new, proper index for that data value. Then, DBTaint not only binds that data value, but the corresponding taint value, if appropriate. This allows the Web application to use parameterized queries with no knowledge of the underlying implementation of the composite types used by DBTaint.

2.4.3 Retrieving Database Values

The results of database queries are tuples of data and taint values. DBTaint extracts the data values from these tuples, then taints them as appropriate in the single-application taint tracking engine used by the Web application. This completes the propagation of taint values back into the Web application.

3 Implementation

We have developed two different prototype implementations of our DBTaint system (one for Perl and one for Java) to demonstrate the effectiveness of our approach. These prototypes are fully capable of working with real-world Web services that use the PostgreSQL database engine.

3.1 Database

Both DBTaint prototypes assume the use of the PostgreSQL database server. PostgreSQL is a popular, full-featured, enterprise-class object-relational database system. Users can create composite types from base types, add custom functions, and overload operators. We leverage these features to manage the taint information stored in our modified database tables.

3.1.1 Composite Types

DBTaint uses *composite types* to store data and taint information in PostgreSQL database tables. A composite type is a type with the structure of a user-defined record, and can be used in place of simple types in the database. Each composite type we create has two elements: the data value, and the associated taint value. We can maintain taint values at whatever granularity we like (e.g. per character) but to simplify our examples here we use a single taint bit. PostgreSQL uses the `ROW()` syntax to specify composite type values, so we express a tainted `INT4` as the `INT4t` composite value `ROW(37, 1)`.

3.1.2 Auxiliary Functions

During initialization, DBTaint determines all the native database types used by the Web application by inspecting the original database tables' schemas. DBTaint uses the `CREATE TYPE` command to create a new PostgreSQL composite type for each of these native types. Before these composite types can be used to create new composite versions of the original database tables, DBTaint creates a number of auxiliary functions to support these new types. These auxiliary functions are used to preserve the behavior expected by the database clients, and to simplify the SQL query processing DBTaint performs at the boundary of the database and other applications.

DBTaint generates the standard comparison operators to allow the database to sort and compare composite type values, and uses PostgreSQL's operator overloading capabilities to add taint-aware capabilities to the common operators (e.g. `>, <=, +, -`). DBTaint creates aggregate functions (using `CREATE AGGREGATE`) in the PostgreSQL database to create taint-aware versions of common aggregate functions, like `MIN` and

MAX. Additionally, DBTaint generates length() functions for composite types with string values, and arithmetic operators for numeric composite types. DBTaint overloads arithmetic operators to provide interoperability with base types, while propagating the taint information to the resulting composite values. For example, the operation "adding the integer 2 to the tuple ROW(5, 1)::INT4t" returns ROW(7, 1)::INT4t, which retains the taint bit from the original tuple. DBTaint also creates getval() and gettaint() functions for the composite types, which extracts just the value or taint bit for a particular piece of data. DBTaint sets the values and taint bits using normal SQL statements, and therefore requires no additional PostgreSQL functions to manipulate this data on the server.

3.1.3 Table Creation

After creating the necessary composite types and auxiliary functions in the database, DBTaint automatically replaces all of the simple types in the database tables with their associated composite type versions. Note that unless a Web application creates new tables during operation, this table creation phase only occurs during the initial installation and configuration stage. DBTaint adapts default values, column constraints, and other table properties as needed to match the new composite types. Default values are considered "untainted" in this process. For example, DBTaint converts a column with type INT4 and default value of 0 into a composite type column of type INT4t with default value ROW(0,0).

3.2 Perl Implementation

We developed an implementation of DBTaint for Web applications written in Perl that use the popular DBI module for accessing PostgreSQL databases. We use a modified version of Perl's "Taint Mode" to perform information flow tracking within the Web application.

3.2.1 Perl Taint Tracking

Our Perl implementation of DBTaint leverages the Perl taint mode to track information flow through the Web application. Perl's taint mode is an active mechanism that prevents some Perl operations from using untrusted user input unsafely. Perl taints user input, and halts when tainted values are used in certain unsafe situations (like as a parameter tosystem). We only needed a passive taint tracking engine for DBTaint, so we provide a modified Perl engine that does not halt in these situations, allowing us to use DBTaint with applications not normally compatible with Perl's taint mode.

3.2.2 Perl DBI

In our Perl implementation we add our DBTaint database interface functionality to the DBI (DataBase Interface) module. The Perl taint mode engine we use in our implementation has a limitation: it only tracks the taint bit of the entire variable as a whole. This means that, for example, a string is either completely tainted, or completely untainted. If a Web application assembles a query string by concatenating tainted and untainted data, by the time this string reaches the database interface it is impossible to determine what parts of the original query was tainted, and what was untainted. Note that this is not a problem if we use a more sophisticated taint tracking engine, such as the one used in our Java implementation below.

But, the Perl taint mode engine is still completely sufficient for DBTaint if the application uses prepared statements for its database queries. Prepared statements are SQL statements with placeholders for parameters to be supplied later. Prepared statements are used for performance reasons, to separate SQL logic from the data supplied, and to help prevent of SQL injection, and are quite common in modern Web applications. When these parameters are supplied later, the DBTaint system can inspect the taintedness of these data values at the database interface. In this way, we can properly propagate the taint information across the boundary of the Web application and the database application.

3.2.3 Other Modifications

The Web application we chose to use with DBTaint used prepared statements for all of its SQL queries, which made the DBI rewriting relatively simple. We slightly modified the Apache-Session Perl module to use prepared statements in a way that matched the rest of the Perl application to simplify our DBI rewriting logic. We also needed to modify the Encode Perl module to avoid user values being inadvertently untainted during conversion from UTF-8 encoding.

3.3 Java Implementation

We developed an implementation of DBTaint for Web applications written in Java that use the popular JDBC API to access PostgreSQL databases. We use a character-level taint tracking system for Java, which allows us to properly rewrite both prepared and non-prepared statements without losing any taint information from the Web application.

3.3.1 Java Taint Tracking

We use a character-level taint tracking engine for Java. [7] This taint engine marks all elements of incom-

ing HTTP requests as tainted (e.g. form parameters, cookies, etc.), and propagates the taint bit throughout the Web application. When these values are passed to the database interface, DBTaint rewrites the queries appropriately to propagate the taint bits between the applications. We were able to use this taint tracking engine without any special configuration or modifications.

3.3.2 JDBC

In our Java implementation we add our DBTaint database functionality to the JDBC (Java DataBase Connectivity) classes. The Java information flow tracking engine we use tracks taint bits on each character of each String object. With this more precise information, we are no longer limited to only prepared statements, as we no longer depend on the parameters being separate from the query to determine if they are tainted or not. When the database interface receives a query with literals embedded in the query string, DBTaint inspects the taint values for the characters of that literal, and then adds the appropriate taint information when tupling the value.

For example, if the DBTaint system receives the following query in the JDBC interface:

```
INSERT INTO messages (msg) VALUES
('first post')
```

DBTaint will inspect the taint values of the substring consisting of `first post`. DBTaint will then rewrite the query with the appropriate taint values based on the taintedness of the substrings. For example, if the `first post` value was tainted, the query would be rewritten to:

```
INSERT INTO messages (msg) VALUES
(ROW('first post', 1))
```

We use Zql [5], a Java SQL parser, to parse the queries so they can be rewritten in DBTaint. Rewriting parameterized queries is performed using the same approach described above in the Perl implementation.

4 Evaluation

To demonstrate the effectiveness of DBTaint in real-world systems, we evaluate the performance of our taint-aware database operations, and run two popular Web services with DBTaint. We executed all benchmarks on a virtual machine running Cent OS 5 on a 2.6 Ghz Intel Core 2 Quad host with 4 GB of RAM. Our DBTaint implementations are based on PostgreSQL version 8.3.7, Perl version 5.10.0, and Java version 6 (1.6).

4.1 Database Operations

We first attempt to evaluate the overhead of the changes we make on the database server. By replacing all primitive data types in the database tables with composite types, the database server now has more information to manage, and is using custom composite types that have not been optimized as thoroughly as the native types. Table 1 contains the average run time of each of the following tasks. Between each run, the database was restarted and cached results were cleared to avoid measuring the effectiveness of the database caches.

Operation	native	DBTaint	overhead
INSERT row	0.5ms	0.6ms	20%
SELECT ALL	23ms	26ms	13%
SELECT WHERE	23ms	26ms	13%
EQUALS op	0.2ms	5ms	2400%
LESS THAN op	0.2ms	2.3ms	1050%
ADDITION op	0.2ms	2.4ms	1100%

Table 1: Database operations incur high overhead (later shown to not dramatically impact overall performance)

We note that the composite versions of many of these operations are a great deal slower than their native counterparts. We hypothesize that these discrepancies are due to the fact that our DBTaint database operations were defined to be simple and portable. The impact of these slower operations in a benchmark of actual Web application performance (Sections 4.5 and 4.6) indicates that the performance penalties paid for more portable implementations of DBTaint may be of little concern in many environments. Furthermore, it may be possible to greatly improve these results by implementing the datatypes and associated functions more efficiently (e.g. in C rather than SQL). We analyze the source of these results in more detail in Section 5.2.

4.2 Web Application: RT

We selected the enterprise-grade ticket tracking Web application named Request Tracker (RT) [1] to evaluate the effectiveness of DBTaint in a realistic environment. RT is not designed to be used with Perl taint mode, and was not created with DBTaint or any other information flow tracking system in mind. It has over 60,000 lines of code. It uses 21 different database tables to store information about tickets entered into the system, users of the system, transaction history of system modifications, access control, and more. Other than installing our composite datatypes and removing the inadvertent untainting in a Unicode conversion function, we ran RT with DBTaint without making any further changes to the Web application. We successfully tracked the flow of user input throughout the entire Web service: from the Web application, into the database, and back.

To demonstrate that DBTaint does not alter the behav-

ior of the Web application, we recorded a series of interactions with the Web application installed in an unmodified environment. We saved the database contents resulting from using the application in the unmodified environment for later reference. Then, after deploying the Web application and running it in the DBTaint system, we replayed these recorded Web actions in this environment. When we compared the values of the database tables in the DBTaint system with the values from the unmodified run, the only differences we observed were expected variances in values like timestamps. We observed that DBTaint allows the Web application to behave exactly as it would in a normal environment, transparently providing the information flow tracking capabilities to the entire Web service.

The RT application was not designed to function in taint mode, and halts immediately if taint mode is enabled in a normal Perl environment. We modified the Perl engine (see Section 3.2.1) to allow RT to function in the taint mode. While we did not use Perl taint mode to prevent active attacks, we analyzed the taint information in the database to learn about information flow through the Web application. Note that if the Web application were designed to function in taint mode, it would not need our modified Perl engine to work with DBTaint.

4.3 Analyzing Database Taint Values

After running RT in DBTaint, we could infer knowledge about the application by simply inspecting the taint values of the data in the database. By glancing through the taint values of the database records, we see that nearly all of the user input stored in the database is marked tainted. This implies that the Web application performs little input filtering, and relies on output filtering to escape characters to prevent XSS attacks. Upon inspection of the application code, we found that the application stores user input directly into the database, and escapes and replaces dangerous characters before displaying them in Web pages.

Columns with only untainted data Many of the database columns contained only untainted values. We observed that the values in these untainted columns were either provided or generated by the Web application, rather than originating from user input. For example, the “type” field of the “tickets” table was always untainted, because these values ranged across only a few hard-coded choices in the RT application. Another column contains a timestamp for internal logging of actions within the database. Because these timestamps were generated by the Web application and not specified by user input, they also appeared untainted in the DBTaint database tables.

Columns with only tainted data There were also columns composed entirely of tainted elements, such as the “subject” column of the “tickets” table. Columns with this property corresponded to mandatory form fields that the user completes while using the application. Because this Web application uses output filtering rather than input filtering, it passed user data directly from the Web application to the database without sanitization. Each element in the column contains untrusted data from user input, and we can immediately tell that the application is not performing input filtering on these values before storing them.

Columns with mixed tainted and untainted data While most table columns contained uniformly tainted or untainted data, there were several columns containing both tainted and untainted data. Upon further investigation, we observe that most of these are the columns for optional form fields. The Web application provides a default (untainted) value, but if the user provides a value of their own, it will show up as tainted in the database. For example, the “finalpriority” column of the “tickets” table has a default value of 0, which is untainted in the database if the user does not specify any value. However, if the user does provide a value it will show up as tainted in the database.

We investigated whether the application might not have sanitized any of these user-supplied values. We discovered that the application always sends data from these columns to the Web framework, which sanitizes the data before outputing them. Even though we did not find any sanitization bugs in RT, DBTaint helped us gain confidence in the completeness of sanitization in RT.

4.4 Enhancing Functionality

While DBTaint can be used to gain insight into the way that data flows throughout the Web application, it can also be used to enhance the functionality of the application without incurring additional security risk. RT escapes angle braces and other potentially dangerous characters from database values before using them to create a HTML page. While this can certainly help prevent cross-site scripting attacks, it also prevents the application from using these dangerous characters in its default values. For example, when a database column contains mixed tainted (user input) and untainted (application default) data, without DBTaint the application must sanitize all of them, unnecessarily restricting default values even though they are safe.

With the cross-application information flow provided by DBTaint, we were able to expand the functionality of the Web application without losing security. Since we can reliably track the flow of tainted data through

Web Application	Overhead
Request Tracker (RT)	12.77%
JForum	8.49%

Table 2: Overall Web service overhead

the Web application and the database, we can avoid concerns of false positives and false negatives that come with single-application taint tracking schemes. Instead of escaping all data values before returning them to a Web visitor, we modified the application to only escape tainted values. Since user data remains tainted through the entire Web service, dangerous characters will be escaped in malicious input. On the other hand, trusted values (such as application defaults) will be untainted and can be safely included in HTML pages without undergoing this same escaping procedure.

4.5 Performance

We tested the impact of DBTaint on the performance of the RT Web application by timing the round trip time of making a request to the Web application, processing the request, and receiving the response. We performed 10 sets of 1,500 requests for the original (unmodified) version of RT, and of RT running with our DBTaint prototype. To simulate and environment of a Web application under load rather than just starting up, we recorded the time for the last 1,250 requests of each set. Recall that our Perl implementation is completely unoptimized, and each SQL query is reprocessed every time the Web application makes a database request. As Table 2 shows, we note that even with no attempt at optimization, we achieve less than 13% overhead in our prototype, which we believe provides a high upper bound of the performance impact of our approach.

4.6 Web Application: JForum

To evaluate the effectiveness of our Java implementation of DBTaint, we selected JForum version 2.1.8, which is (according to the documentation) a "powerful and robust discussion board system." [2] JForum includes more than 30,000 lines of code in 350 Java classes, and uses 59 database tables to maintain subforums, posts, messages, access control, and more. We deployed JForum to a Tomcat server with the character-based taint tracking engine described in Section 3.3.1.

We evaluated our Java implementation of DBTaint in a similar way to our Perl evaluation in Section 4.2. We recorded a series of Web events including logging in, posting to the forum, and viewing existing posts. We determined the performance overhead of our Java implementation on JForum to be less than 9% (Table 2).

Because our Java implementation uses a character-based taint tracking engine, the query rewriting phase is more sophisticated and complex than our Perl implementation. This is because it handles both parameterized and non-parameterized queries, and checks the taint values of each character in data strings. With this approach, we originally observed an overhead of close to 30%. However, in our Java implementation, we added very simple memoization to the parsing and rewriting of parameterized queries, which dropped the performance overhead to less than that of the Perl implementation, despite the increased complexity. In situations where Web services serve far more requests than there are distinct parameterized queries (which we believe is the common case), caching the results from the query rewriting phase is a simple way to improve performance in implementations with sophisticated character-level query rewriting analysis.

5 Discussion

DBTaint is an effective system for providing cross-application information flow tracking through databases. In this section we outline some of the benefits of DBTaint over other systems, reflections on our prototype implementations, and applications of DBTaint to interesting security problems.

5.1 Benefits

DBTaint has the following major benefits:

- End-to-end taint tracking throughout all applications in a Web service.
- Full support of the semantics of database operations. DBTaint tracks taint flow at the high database operational semantics level, rather than at the low instruction level.
- Efficiency. DBTaint only tracks the information flow within the database and between the database and its client applications, avoiding the overhead of the extra tracking that system-wide solutions perform. Our unoptimized prototypes add only a minor performance penalty to Web services.
- Only SQL-level changes to the database server, and no changes to the Web application. Our major implementation work is in modifying the database interface. We don't need to make any changes to the database client because DBTaint intercepts and automatically rewrites all SQL queries from the client as needed for our information flow tracking.

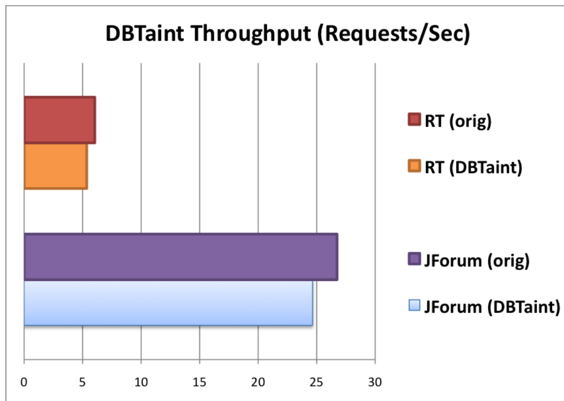

Figure 1: DBTaint Serial Throughput

5.2 Database Performance

The performance impact of the composite forms of database operations (summarized in Table 1) may appear to be surprising at first. We note that the composite versions of `INSERT` statements execute somewhat more slowly than the original queries. This is not particularly surprising, as the database client is inserting twice as many values in the composite version (a taint value for each data value). Similarly, basic `SELECT` statements are slightly slower than the original queries, likely for similar reasons – there is simply more data to work with in the composite version.

However, we note that the other operations (the equality operator, the less than operator, and the addition operator) are *much* slower than their native counterparts. We suspect that this is because native types have been highly optimized in the underlying database engine. In contrast, we added the composite version of these operators using high-level SQL functions. We hypothesize that while most Web application performance is not bound by mathematical operators in the database, substantial performance improvements could be made by implementing these composite types and their associated functionality in C rather than SQL. These optimized datatypes could be dropped in as replacements for our prototype SQL implementation if the extra performance was necessary. However, in testing our implementations with real-world Web applications (Sections 4.5 and 4.6) we observed that despite the large performance overhead of these operations, the overall performance of the Web application was not dramatically impacted. As shown in Figure 1, DBTaint has relatively little impact on the throughput of the Web application serving requests despite the overhead of database operations.

5.3 Applications of DBTaint

Persistence of taint information DBTaint allows the taint information stored in the database to remain persistent through multiple runs of the applications. This allows an application that uses the database to run many times without losing the taint information from the previous executions.

Comparison of different versions of application DBTaint can be used to compare two different versions of an application that use a database for storage. After refactoring some user input sanitization code, for example, programmers can run the old and new versions of a Web application under DBTaint and compare the resulting database tables. Variations in the taint patterns of the database columns may indicate a change in input sanitization policies.

Identification of incomplete input sanitization Incomplete input sanitization contribute to many security vulnerabilities. Common solutions for detecting incomplete input sanitization are static analysis and runtime testing. Static analysis techniques are often expensive and are prone to false positives and negatives. For runtime testing, the testers must understand the sanitization functions of the program to design malicious input to test the completeness of the sanitization functions. DBTaint provides an alternative mechanism to detect incomplete input sanitization at runtime and requires no understanding of the sanitization functions in the program.

Web applications typically sanitize untrusted input at two moments: (1) input filtering, which sanitizes an input as it enters the program; and (2) output filtering, which sanitizes untrusted data just before the program embeds the data into a generated Web page.

DBTaint has the ability to provide immediate insight into the taint properties of the data in an application without requiring the user to understand the application code. For Web applications that perform no input filtering,

- Columns that contain only tainted data are likely for storing mandatory user data.
- Columns that contain only untainted data are likely for storing data hardcoded from or generated by the applications themselves, rather than user input.
- Columns that contain mixed taint and untainted values are used for multiple purposes (e.g. data coming from different applications or code paths), or for optional data fields whose default value (set by the program) is untainted but user value is tainted.

In applications that perform input filtering, such column analysis can be even more useful.

- Columns where data are completely untainted indicate that all of the values are either sanitized user input or values produced by the application.
- Columns where data are completely tainted suggest user data that has not been properly sanitized, which may indicate a security vulnerability.
- Columns containing both tainted and untainted data may indicate that the input sanitization is incomplete, i.e., the program sanitizes input data on some paths but not on the other paths.

In the last two cases, DBTaint helps the auditor to reduce the search space for potential sanitization bugs.

5.4 Inadvertent Untainting

When using a taint engine, one must be careful to never inadvertently untaint tainted data. DBTaint does not untaint any tainted values (manually or automatically), but unfortunately the two single-application taint tracking engines we used for our Perl and Java prototypes did perform some inadvertent untainting. This is not the fault of DBTaint, but the problems in these other engines did make our evaluation more difficult.

Perl's taint mode is designed to automatically remove the taint bit on data when it is matched against a regular expression. Perl assumes that a programmer using a regular expression on a variable is validating the contents of the variable, so Perl automatically untaints the value. However, some Perl application and library code uses regular expression for simple string processing, rather than validation or sanitization, leading to inappropriate untainting. We discovered that an encoding/decoding UTF-8 conversion function was untainting all user input in the RT application before the data reached the DBTaint database interface. We addressed this problem by manually retainting the results of the function when the original string was tainted before the decoding.

We encountered similar difficulties using our character-based taint tracking engine for Java. The Java engine we used provides efficient taint tracking by extending String and other String-based classes to maintain taint data. However, because the primitive data types are not similarly extended, the engine cannot track taint bits for a String converted to a character array and back, for example. All taint bits are lost, inadvertently and incorrectly untainting the resulting String. Due to this limitation, some of the tainted values from the JForum Web application received via POST submissions became untainted before they reached the DBTaint database interface. As some user input values were inadvertently untainted, we were unable to perform a meaningful analysis of the taint values of each database column.

6 Related Work

The ability to access a Web service from anywhere means that it must be able to handle input from any source. Unchecked malicious input can lead to some of the top reported software vulnerabilities in Web applications [3]. The information flow tracking provided by DBTaint is like a coarse-grained version of where-provenance [6], allowing developers to identify unchecked user input though multiple applications in the Web service without requiring a whole-system solution.

Application-wide information flow tracking Splint [9] supports source code annotations that help a programmer identify the flow of tainted information within a program. TaintCheck [16] identifies vulnerabilities automatically by performing this analysis on a binary running within its own emulation environment. Xu et al. [19] leverage the source code of a program produce a version of that program that can efficiently track information flow and identify attacks. WebSSARI [12] targets web applications written in PHP specifically with static analysis to identify the information flow of unvalidated input and adds runtime sanitization routines to potentially vulnerable code using that input. Lam et al. [13] also targets web vulnerabilities with the automatic generation of static and dynamic analyses of a program from a description of an information flow pattern. Because most modern Web services include multiple applications, single-application information flow tracking systems result in false positives and/or negatives because they must assume database values are either tainted or untainted without complete runtime information. DBTaint avoids any need for manual annotation and automatically provides information flow propagation across Web service applications.

System-wide information flow tracking With architectural support, information flow tracking systems can trace untrusted I/O throughout the system [17, 8] at a fine memory address level granularity. These systems however, require substantial changes to the underlying hardware or must emulate the entire system with a performance penalty. Ho et al [11], provide the same system-wide tracking with the Xen virtual machine monitor and switch to a hardware emulator only when needed to mitigate the performance penalty. However, these approaches pay an unnecessary performance cost by tracking much more than necessary for most Web services. Other system-wide information flow tracking systems like HiStar [20] and Asbestos [18] are too coarsely grained to track taint values of individual values throughout the Web application and the database. These system-wide approaches also fail to take advantage of the se-

mantics of information flow during database operations. WASC [15] targets web applications and provides a dynamic checking compiler to identify flows and automatically instrument programs with checks. They add support for inter-process flow tracking through databases by maintaining external logs of all SQL transactions, operands and associated tags. This approach lacks DBTaint's ability to use taint values during internal database operations (e.g. preferring untainted values in equality operations for `SELECT DISTINCT` queries).

7 Conclusion

We have designed and implemented DBTaint, which provides information flow tracking in databases to enable cross-application information flow tracking. When database clients, such as Web applications, write into the database, DBTaint stores data together with their taint information. When the database clients retrieve data, DBTaint tags the data with proper taint information. Our implementation requires no modification to Web applications, and only SQL-level additions to the database. By interposing on the database interfaces between Web applications and databases, DBTaint is transparent to Web applications. We demonstrated how two Web applications, an enterprise-grade application written in Perl (RT) and a robust forum application written in Java (JForum), easily work with DBTaint. DBTaint not only can enable cross-application taint tracking but may also identify potential security vulnerabilities due to incomplete sanitization without the need to understand sanitization functions in the Web application. Because DBTaint takes advantage of the semantics of database operations, its overhead is low, and our unoptimized prototype implementations add only 10-15% overhead to the entire system.

8 Acknowledgment

We wish to thank David Wagner and Erika Chin for helpful discussions and for providing Java character-level taint tracking.

References

[1] Best Practical: Request Tracker. `http://bestpractical.com/rt/`.

[2] JForum. `http://jforum.net/`.

[3] OWASP top 10 2007. `http://www.owasp.org/index.php/Top_10_2007`.

[4] SANS: Top 20 internet security problems, threats and risks. `http://www.sans.org/top20/`.

[5] Zql: Java SQL Parser. `http://www.gibello.com/code/zql/`.

[6] BUNEMAN, P., KHANNA, S., AND TAN, W. C. Why and where: A characterization of data provenance. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory* (London, UK, 2001), Springer-Verlag, pp. 316–330.

[7] CHIN, E., AND WAGNER, D. Efficient character-level taint tracking for java. In *SWS '09: Proceedings of the 2009 ACM workshop on Secure web services* (New York, NY, USA, 2009), ACM, pp. 3–12.

[8] CRANDALL, J. R., WU, S. F., AND CHONG, F. T. Minos: Architectural support for protecting control data. *Transactions on Architecture and Code Optimization 3* (2006), 359–389.

[9] EVANS, D., AND LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE Software* (2002).

[10] GUNDY, M. V., AND CHEN, H. Noncespaces: using randomization to enforce information ow tracking and thwart cross-site scripting attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2009), pp. 1–18.

[11] HO, A., FETTERMAN, M., CLARK, C., WAR, A., AND HAND, S. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (2006), pp. 29–41.

[12] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D. T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web* (2004), pp. 40–51.

[13] LAM, M. S., LIVSHITS, B., AND WHALEY, J. Securing web applications with static and dynamic information flow tracking. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation* (2008).

[14] NADJI, Y., SAXENA, P., AND SONG, D. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium* (2009).

[15] NANDA, S., LAM, L.-C., AND CHIUEH, T.-C. Dynamic multi-process information flow tracking for web application security. *ACM/IFIP/USENIX 8th International Middleware Conference (Middleware'07)* (2007), 1–20.

[16] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2005).

[17] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2004), ACM, pp. 85–96.

[18] VANDEBOGART, S., EFSTATHOPOULOS, P., KOHLER, E., KROHN, M., FREY, C., ZIEGLER, D., KAASHOEK, F., MORRIS, R., AND MAZIÈRES, D. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst. 25*, 4 (2007), 11.

[19] XU, W., BHATKAR, S., AND SEKAR, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. *Usenix Security 2006* (2006).

[20] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 263–278.

Notes

[1] We have not yet overcome the bugs in the original PHP taint implementation, which crash the PHP interpreter.