Defending Against Injection Attacks Through Context-Sensitive String Evaluation

Tadeusz Pietraszek[1] and Chris Vanden Berghe[1,2]

[1] IBM Zurich Research Laboratory,
Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland
[2] Katholieke Universiteit Leuven,
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{pie, vbc}@zurich.ibm.com

Abstract. Injection vulnerabilities pose a major threat to application-level security. Some of the more common types are SQL injection, cross-site scripting and shell injection vulnerabilities. Existing methods for defending against injection attacks, that is, attacks exploiting these vulnerabilities, rely heavily on the application developers and are therefore error-prone.

In this paper we introduce CSSE, a method to detect and prevent injection attacks. CSSE works by addressing the root cause why such attacks can succeed, namely the ad-hoc serialization of user-provided input. It provides a platform-enforced separation of channels, using a combination of assignment of metadata to user-provided input, metadata-preserving string operations and context-sensitive string evaluation.

CSSE requires neither application developer interaction nor application source code modifications. Since only changes to the underlying platform are needed, it effectively shifts the burden of implementing countermeasures against injection attacks from the many application developers to the small team of security-savvy platform developers. Our method is effective against most types of injection attacks, and we show that it is also less error-prone than other solutions proposed so far.

We have developed a prototype CSSE implementation for PHP, a platform that is particularly prone to these vulnerabilities. We used our prototype with phpBB, a well-known bulletin-board application, to validate our method. CSSE detected and prevented all the SQL injection attacks we could reproduce and incurred only reasonable run-time overhead.

Keywords: Intrusion prevention, internal sensors, injection attacks, web applications, PHP.

1 Introduction

In recent years we have seen a steady increase in the importance of application-level security vulnerabilities, i.e., vulnerabilities affecting applications rather than the operating system or middleware of computer systems. Among application-level vulnerabilities, the class of *input validation* vulnerabilities is the most prominent one [11] and deserves particular attention.

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 124–145, 2006.

Input validation vulnerabilities are flaws resulting from implicit assumptions made by the application developer about the application input. More specifically, input validation vulnerabilities exist when these assumptions can be invalidated using maliciously crafted input to effect a change of application behavior that is beneficial to the attacker.

Different types of input validation vulnerabilities exist, depending on the invalid assumption. Buffer overflow vulnerabilities result from invalid assumptions on the maximum size of the input. Integer overflow attacks result from invalid assumptions on the range of the input. Similarly, injection vulnerabilities result from invalid assumptions on the presence of syntactic content in the application input. This work focuses on this last class of vulnerabilities and the attacks exploiting them. In these attacks, the so-called injection attacks, the attacker provides maliciously crafted input carrying syntactic content that changes the semantics of an expression in the application. The results are application-dependent, but typically lead to information leakage, privilege escalation or execution of arbitrary commands.

This paper introduces Context-Sensitive String Evaluation (CSSE), which is an intrusion detection and prevention method, for injection attacks. It offers several advantages over existing techniques: it requires no knowledge of the application or application source code modifications and can therefore also be used with legacy applications. It is highly effective against most types of injection attacks, not merely the most common ones. It does not rely on the application developer, which makes it less error-prone. Finally, it is not tied to any programming language and can be implemented on a variety of platforms.

CSSE effectively shifts the burden of implementing countermeasures against injection attacks from the many application developers to the small team of security-savvy platform developers. This is analogous to the way, for example, the Java platform removed the burden of bounds checking from the application developers, thereby making applications for the Java platform virtually immune to buffer-overflow attacks. CSSE requires a separate implementation for every platform one wants to protect. However, as the number of platforms is several orders of magnitude smaller than the number of applications running on them, these implementations can be implemented by security professionals and undergo thorough testing.

The contribution of this paper is twofold. First, it presents a unifying view of injection vulnerabilities, which facilitates reasoning about this class of vulnerabilities and predicting new types of related vulnerabilities. Second, and central to this paper, it introduces CSSE as a method for defending against injection attacks by addressing the root cause of the problem.

The paper is structured as follows. The next section discusses injection vulnerabilities and the conditions that enable them. Section 3 gives an overview on related work. In Section 4 we provide a detailed overview of CSSE. Our CSSE prototype implementation for PHP is discussed in Section 5. In Section 6 we present experimental results on the effectiveness and efficiency of our implementation. Finally, in Section 7, we present our conclusions and future work.

2 Injection Vulnerabilities

We introduce this class of vulnerabilities with a simple example that is vulnerable to both SQL and shell injections. Next, we identify the root cause or underlying reason why these vulnerabilities are present in applications. Finally, we present a unifying view of the different types of injection vulnerabilities.

2.1 Important Properties of Injection Vulnerabilities

Injection vulnerabilities are programming flaws that allow an attacker to alter the semantics of an expression in an application by providing input containing syntactic content. In this section we give an example of code with SQL-injection and shell-injection vulnerabilities to discuss some of the important properties of these vulnerabilities.

The code below shows a realistic example of a part of a PHP application, responsible for authentication by means of an e-mail address (`$email`) and a numeric pincode (`$pincode`) against credentials stored in a database. The user is successfully authenticated if a non-empty result set is returned.

```
$query = "SELECT * FROM users WHERE email='" . $email . "' AND pincode=" .
    $pincode;
$result = mysql_query($query);
```

This code is prone to several SQL injection attacks. If the attacker provides "`alice@host' or '0'='1`" (note the use of quotes) as e-mail address, the application executes a query, whose result is independent of the pincode provided. Because of operator precedence, such a query will be equivalent to the one with a single condition "`email='alice@host'`", thus allowing the attacker to bypass the authentication logic. Similar attacks executed using the pincode variable, which is used in a numeric context, do not require single quotes in the user input. For example, by using a valid e-mail address (e.g., "`alice@host`") and "`0 or 1=1`" as a pincode, the attacker would again be able to authenticate without proper credentials.

Continuing with our example to demonstrate a shell injection, the code shown below sends a confirmation email to an email address provided.

```
$query = "SELECT * FROM users WHERE email='" . $email . "' AND pincode=" .
    $pincode;
$result = mysql_query($query);
```

In this case, any of the shell metacharacters (e.g., `‘`, `&&`, `;`, newline) in the e-mail address field can be used to execute arbitrary commands on the server. For example, if the attacker uses "`alice@host && rm -rf .`" as e-mail address, the webserver would, in addition to sending an e-mail, try to remove all files from the current directory.

In all our examples, maliciously crafted input carries *syntactic content*. Content is considered syntactic, when it influences the form or structure of an expression. This change of structure ultimately results in altered expression semantics. Which characters qualify as syntactic content depends on the context in which the expression is used (e.g., SQL or shell command). Moreover, the context also

depends on how the input is used within the expression (e.g., string constant vs. numeric pincode in an SQL statement in our example). Identifying all syntactic content for the different contexts is thus a major challenge.

Removing single quotes and spaces from the input would prevent the attacks we described, but would certainly not fend off all attacks. Other dangerous characters include comment sequences (`--`, `/*`, `*/`) and semicolons (`;`), but also this list is not exhaustive [8].

Moreover, database servers commonly extend the ANSI SQL standards with proprietary features and helpfully correct small syntactic errors, e.g., allow the use of double quotes (`"`) instead of single quotes (`'`) for delimiting string contants. As the necessary checks are database-specific, an application can become vulnerable by a mere change of the database backend.

2.2 The Root Cause

Injection vulnerabilities are commonly classified as input validation vulnerabilities. However, the example of Section 2.1 suggests that validating user input to prevent these attacks is nontrivial and error-prone. Treating these vulnerabilities as mere input validation vulnerabilities is therefore an oversimplification.

Instead, we should address their *root cause*, which can potentially yield a less error-prone and more stable solution. Finding this root cause is equivalent to unveiling the underlying reason why a vulnerability is present in a specific system. In the case of vulnerabilities leading to injection attacks, this means determining why specially crafted user input can be used to change the semantics of an expression in the first place.

A common property of injection vulnerabilities is the use of textual representations of output expressions constructed from user-provided input. *Textual representations* are representations in a human-readable text form. *Output expressions* are expressions that are handled by an external component (e.g., database server, shell interpreter).

User input is typically used in the data parts of output expressions, as opposed to developer-provided constants, which are also used in the control parts. Therefore, user input should not carry syntactic content. In the event of an injection attack, specially crafted user input influences the syntax, resulting in a change of the semantics of the output expression. We will refer to this process as *mixing of control and data channels*.

Injection vulnerabilities are not caused by the use of textual representation itself, but by the *way* the representation is constructed. m Typically user-originated variables are serialized into a textual representation using string operations (string concatenation or string interpolation, as in our example). This process is intuitively appealing, but ultimately *ad hoc*: variables loose their type information and their serialization is done irrespectively of the output expression. This enables the mixing of data and control channels in the application, leading to injection vulnerabilities.

We thus consider the *ad-hoc serialization of user input* for creating the textual representation of output expressions as the root cause of injection attacks.

Ad-hoc serialization of user input (or variables in general) can lead to the undesired mixing of channels, but has also some desirable properties. The most important is that it is intuitively appealing and, consequently, more easily written and understood by the application developers. Second, for many types of expressions (e.g., XPath, shell command) ad-hoc serialization of user input using string operations is the only way of creating the textual representation.

Considering this, a defense against injection attacks should enable the application developer to use the textual representation in a safe manner. CSSE achieves this through a platform-enforced separation of the data and control channels, thereby addressing the root cause of injection vulnerabilities, while at the same time maintaining the advantages of textual representation and ad-hoc serialization of user variables. We present the method in more detail in Section 4.

2.3 A Unifying View of Injection Vulnerabilities

Section 2.1 introduced some of the more common types of injection vulnerabilities, but several others exist. In this section we provide a unifying view of the different types.

For any type of injection vulnerability to be present in an application, two prerequisites need to be met. The first is that the application has to use an output expression created using ad-hoc serialization of variables. The second is that the output expression depends on user-provided input data, so it can be influenced by the attacker. Hereafter, we will use the terms *input vector* and *output vector* to refer to classes of input sources and output expressions, respectively.

In Table 1 we categorize some known examples of injection vulnerabilities according to their input and output vectors, and provide a CAN/CVE [10] number if available. The cells in the table show possible avenues for different types of injection vulnerabilities.

The rows of the table represent three coarse-grained categories of input vectors: *network input, direct input and stored input.* Network input consists of all input provided by remote users. It is a combination of transport-layer input (e.g., POST data and cookies in HTTP), and application-level input (e.g., a SOAP request). Direct input, on the other hand, is input that is passed on via a local interface, e.g., through a command-line interface or environment variables. Finally, stored input is input that does not come from the user directly, but involves an intermediate storage step, e.g., in a database or an XML file. Note that for some applications the distinction between network input and direct input may not be clear-cut (e.g., CGI applications access HTTP request data through environment variables). We nonetheless distinguish between these types as they most often use different programming interfaces.

The columns of the table represent the output vectors or types of expressions to be handled by the external components. We distinguish between the following categories: *execute, query, locate, render and store.* The "execute" category covers expressions containing executable content, such as shell commands or PHP scripts. The "query" category contains expressions that are used to select and manipulate data from a repository, e.g., XPath, SQL or regular expressions. The

Table 1. Examples of different injection vulnerabilities with their CVE/CAN numbers. The most common vulnerability types are marked in **bold**.

Input \ Output	Execute (e.g., shell, XSLT)	Query (e.g., SQL, XPath)	Locate (e.g., URL, path)	Render (e.g., HTML, SVG)	Store (e.g., DB, XML)
Network input (GET/POST)	**shell inj.** (CAN-2003-0990)	**SQL inj.** (CVE-2004-0035)	**path traversal** (CAN-2004-1227)	**"phishing" through XSS** (CAN-2004-0359)	**preparation for n^{th}-order inj.**
Direct input (arguments)	**command inj.** (CAN-2001-0084)	regexp inj.	local path traversal	PostScript inj. (CAN-2003-0204)	
Stored input (DB, XML)		n^{th}-order SQL inj.		**XSS** (CAN-2002-1493)	preparation for $(n+1)^{th}$-ord. inj.

"locate" category is related to the previous one, but contains expressions that help locating the repositories themselves, e.g., paths and URLs. Expressions in the "render" categories contain information about the visualization of the data, e.g., HTML, SVG and PostScript. Finally, the "store" category consists of expressions for storing data in a repository. This last category is special as the cells of this column do not represent injection vulnerabilities, but rather the "preparation" for higher-order injections.

Such higher-order injection are defined as injections in which the input inflicting the injection is saved in a persistent storage first. The actual injection only happens when this stored data is being accessed and used. Well-known examples of second-order injections are SQL injections and XSS, where stored data, used in the creation of SQL queries and HTML output, is interpreted as a part of SQL and HTML syntax, respectively. Attacks higher than second-order are less common, but potentially more dangerous, as persistent data is usually considered more trusted. Note that our definition of higher-order injection is broader than that by Ollmann [13], which emphasizes its delayed nature. In our approach, we focus on its basic characteristics, that is, the persistent storage of offending data regardless whether its effect is immediate (as with some XSS attacks) or not (as with the attacks shown by Ollmann).

The table provides a unifying view of all types of injection vulnerabilities. We can use it to classify existing vulnerabilities, but it also provides insight into vulnerabilities that we expect to appear in the future. For example, although we have not yet seen any XPath injection vulnerabilities, it is likely that we will see them appear as the underlying technologies become widely used. It also shows that some vulnerabilities that typically are not regarded as injection vulnerabilities, e.g., path traversal, are in fact very much related and can be prevented using the same techniques as for other injection vulnerabilities.

Figure 1 shows the dataflow in an application from the perspective of this paper. The data flows from multiple inputs and constants through a chain of string operations to form the output expressions. The dashed lines depict the example of Section 2.1 where a single input can result in different outputs depending on the path in the flow graph. The difficulty of securing such an application lies in the fact that all the possible paths between inputs and outputs have to be taken into account.

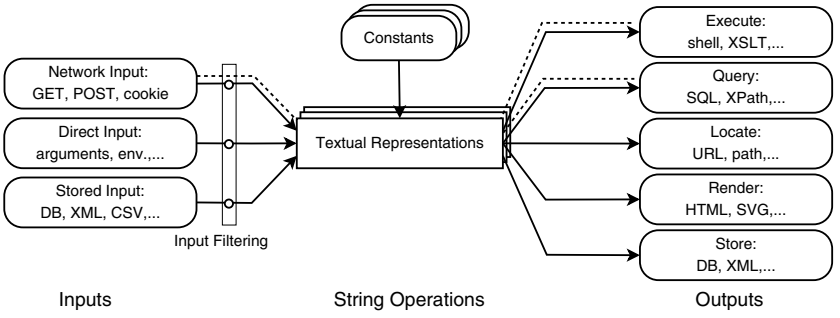

Fig. 1. Use of textual representation in an application. The dashed lines depict the example of Section 2.1, where a single input can result in different outputs.

As an example, web applications have typically many different input vectors: GET and POST parameters, URL, cookies, authentication information and other HTTP headers for every possible request. Moreover, input can come from databases, XML files or other external sources. At the same time, a typical web application can have several output vectors: HTML output for every page that can possibly be generated, a database or XML file, e-mails, etc.

The large number of combinations makes adding the necessary input validation checks error-prone. This is not only true for web applications, but other programs handling user input are also affected. However, for various reasons, web applications tend to be particularly vulnerable. They are typically text-based, are often constructed from loosely-coupled components and exist in a hostile environment, where they receive a lot of untrusted user input. In addition, there is often a lack of proper development tools and the security aspects are not the main focus of the application developers.

What both Table 1 and Figure 1 cannot show is the impact a certain injection vulnerability can have on the security of an application. For example, the SQL injection in Section 2.1 leads to the possibility of authentication without proper credentials. In other cases, an injection results in run-time errors, confidentiality or integrity problems. The actual impact is thus highly situation-specific.

3 Related Work

The prevalence of attacks exploiting buffer-overflow vulnerabilities motivated a considerable research effort focused on preventing and detecting these vulnerabilities. Considerably less attention has been given to the related problem of injection vulnerabilities [3], which instead has been investigated mainly by practitioners [1, 2, 8, 12, 13]. We distinguish between two coarse-grained categories of existing solutions: "safe ad-hoc serialization" and "serialization APIs". In this section we present them and discuss their advantages and disadvantages.

Safe Ad-Hoc Serialization. The first category contains solutions facilitating safe ad-hoc serialization. *Manual input validation* falls into this category, and because

of its conceptual simplicity it remains the most popular approach. It involves manually checking all the user-provided input for syntactic content, which will then be escaped or rejected. Tool support, typically in the form of an API, is limited to predefined filters for certain output vectors. The use of these filters remains the responsibility of the application developer.

Manual input validation is error-prone as it heavily relies on the application developer. Implementing it correctly proves very difficult because of the following issues. First, applications often have a large number of inputs and the handling code is scattered throughout. Manually capturing all these inputs can therefore be a daunting task. In addition, the checks are by necessity highly dependent on the context and can be very complex for particular output vectors. Finally, care has to be taken that the checks are performed at the right place, as checks performed before the variable is in its final encoding may lead to encoding vulnerabilities. These exist when the validation can be foiled using special encoding, e.g., with entities in XML or URL encoding.

Automated input validation is a second approach, which aims at making input validation less error-prone by not relying on the application developer. The best known example is "MagicQuotes" in PHP [14], which operates by validating all input data at the time it is received. The second issue we raised for manual input validation applies also to this approach, as the usage context is not fully known when the validation is performed. Consequently, it is not defined what content should be considered syntactic. Instead, common output vectors are assumed and the validation is performed accordingly. This can lead to vulnerabilities when the assumption proves incorrect.

Variable tainting in Perl [20] is a third approach, addressing the first issue of manual input validation, namely the large number of inputs scattered throughout the code. It works by "tainting" all input to the application and warning when dependent expressions are used without having undergone manual validation. The application developer still is responsible for implementing the actual checks, but the tainting mechanism makes it less likely that necessary checks are overlooked. Tainting of input variables, inspired by Perl, has also been applied to other languages for preventing buffer overflows. Larson and Austin [6], instrument string operations in C programs to find software faults caused by improperly bounded user input. Shankar et al. [17] use static taint analysis to detect format string vulnerabilities in the compile phase.

The last approach in this category is provided by SQLrand [3], which prevents SQL injections by separating commands encoded in the program code from user-supplied data. SQLrand is based on the assumption that syntactic parts of SQL commands can only appear as constants in the program code and should not be provided by user input. SQLrand preprocesses the source code of applications and replaces all SQL commands with encoded versions. The modified commands are then intercepted by an SQL proxy, which enforces that only correctly encoded instructions are passed on to the database. The main disadvantages of this approach are that it requires a complex setup and that it is specific to SQL.

Serialization APIs. The second category consists of solutions that can be characterized as serialization APIs (Application Programming Interfaces). These APIs assist the application developer in serializing variables and thus creating a safe textual representation. They either do not use explicit textual representation at all, and the representation is created using a programmatic API instead, or they use special serialization templates, in which the textual representation is created by the application developer and only the variables are serialized using an API. An example of the former type is DOM (Document Object Model), which provides programmatic support for creating XML documents, thereby, in addition to its other advantages, preventing XML injection attacks. Examples of the latter type include serialization templates for SQL, which exist for many different programming languages: `PreparedStatement` in Java, ADOdb [7] in PHP and Python, `SQLCommand` in VisualBasic and DBI [4] in Perl.

The key advantage of this approach is that the serialization is handled automatically by the platform. Although the method is less error-prone, some problems remain. First, the tool support is limited to some frequently used output vectors. For example, there are prepared statements for SQL expressions and DOM for XML, but we know of no similar tool support for XPath or regular expressions. Second, the application developer still is responsible for actively and correctly using this mechanism. And third, there is a large number of legacy applications that do not use this functionality or run on platforms that do not provide this tool support.

Also in this category is the approach taken by Xen [9], which fully integrates XML and SQL with object-oriented languages, such as C♯. Xen extends the language syntax by adding new types and expressions, which avoids ad-hoc serialization and thus prevents injection vulnerabilities. The disadvantage of this method is that it cannot be easily applied to existing applications.

4 Context-Sensitive String Evaluation

In this section we provide a detailed description of CSSE and show how it compares to the existing methods for defending against injection attacks.

CSSE addresses the root cause of injection vulnerabilities by enforcing strict channel separation, while still allowing the convenient use of ad-hoc serialization for creating output expressions. A CSSE-enabled platform ensures that these expressions are resistant to injection attacks by automatically applying the appropriate checks on the user-provided parts of the expressions. CSSE achieves this by instrumenting the platform so that it is able to: (i) distinguish between the user- and developer-provided parts of the output expressions, and (ii) determine the appropriate checks to be performed on the user-provided parts.

The first condition is achieved through a tracking system that adds metadata to all string fragments in an application in order to keep track of the fragments' origin. The underlying assumption is that string fragments originating from the developer are trusted, while those originating from user-provided input are untrusted. The assignment of the metadata is performed without interaction of the application developer or modification of the application source code, and

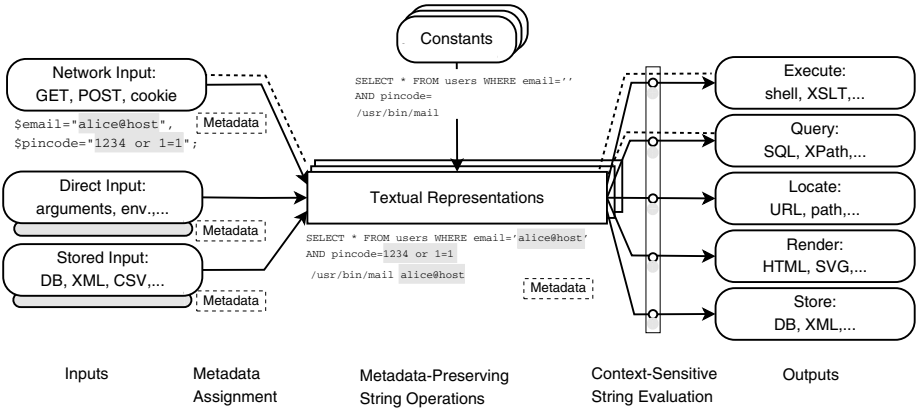

Fig. 2. Using CSSE to preserve the metadata of string representations and allow for late string evaluation. Shades represent string fragments originating from the user.

is instead achieved through the instrumentation of the input vectors (e.g., network, file) of the CSSE-enabled platform. CSSE further instruments the string operations to preserve and update the metadata assigned to their operands. As a result, the metadata allows us to distinguish between the developer-provided (trusted) and user-provided (untrusted) parts of the output expressions at any stage in their creation. Figure 2 illustrates the dataflow of the vulnerable application executed in a CSSE-enabled platform.

The second condition is achieved by deferring the necessary checks to a very late stage, namely up to the moment when the application calls the API function to pass the output expression on to the handling component (output vector). CSSE intercepts all API calls related to output vectors, and derives the type of output vector (e.g., MySQL, shell) from the actual function called (e.g., `mysql_query()`, `exec()`). This allows CSSE to apply the checks appropriate for this particular output vector.

At this point, CSSE knows the complete context. The first part of the context is provided by the metadata, which describes the fragments of the output expression that require checking. The second part of the context is provided by examining the intercepted call to the API function, which determines which checks will be executed. CSSE then uses this context information to check the unsafe fragments for syntactic content. Depending on the mode CSSE is used in, it can escape the syntactic content or prevent the execution of the dangerous content (both intrusion prevention) or raise an alert (intrusion detection).

The novelty of our method lies in its ability to automatically gather all the required pieces of information that allow it to perform the necessary checks for detecting and preventing injection vulnerabilities. A CSSE implementation is platform-specific, but effective for all applications executed on this platform. No analysis or modification of the application is required, except for very rare cases where user-provided input is explicitly trusted. This will be further discussed in the remainder of this section.

CSSE compares favorably to the existing methods described in Section 3. Because its checks are platform-enforced and performed when the expression is already encoded, it has none of the disadvantages that make the existing safe ad-hoc serialization methods error-prone. It also has several advantages over serialization APIs, as it is applicable to a wide variety of output vectors, requires no application developer actions and can also be used on legacy applications.

In the remainder of this section, we describe four logical parts that make up a CSSE implementation: metadata representation, metadata assignment, metadata-preserving string operations and context-sensitive string evaluation. The first three form the metadata tracking system, whereas the last part is responsible for determining and executing the appropriate checks. Here, we focus on the architectural aspects; the implementation will be discussed in Section 5.

Metadata Representation. In CSSE, the term "metadata" refers to information about the origin (user-provided or developer-provided) of all the fragments that make up string variables. Conceptually, this metadata is attached to the string variables, as it travels with them through the application.

However, the actual implementation of the metadata is highly platform-dependent. For example, the metadata can be stored either in a platform-wide repository or indeed as part of the actual data object. Also, the metadata itself can be represented in several ways, e.g., using a bitmap or a list of pointers delimiting different parts of an expression. Finally, the absence of metadata for a variable can also implicitly carry information on its origin.

CSSE metadata is similar to variable taint in Perl, as it also denotes the origin of the string variables and thus whether they are trusted or untrusted. However, for our method a richer metadata representation is needed. While variable taint in Perl only describes if there exists a fragment of the string variable originating from the user, CSSE metadata describes the origin of *all* the individual fragments that make up a string variable (cf. shaded fragments in Figure 2).

It is also possible to use the CSSE metadata to track a "history" of the data, by keeping track of the chain of operations performed on its fragments (e.g., filtering, escaping quotes) to ensure that the validation method applied is appropriate to the output vector (e.g., checking for database metacharacters is inappropriate when the variable is used as a part of a shell command). However, in the remainder of the paper we limit the scope of the metadata to describing origin, as this is sufficient for our purposes.

Metadata Assignment. A CSSE-enabled platform automatically assigns metadata to all string variables. For user-provided input, this is achieved through the instrumentation of the input vectors of the platform. When the application receives input from the user, e.g. in the form of an HTTP parameter, the instrumented platform API will ensure that the received variables are provided with the appropriate metadata, marking them untrusted. On the other hand, static string constants present in the program code are automatically considered safe. There is no need for the application developer to modify them or anyhow indicate how they will be used (e.g., as an SQL or shell command, HTML code).

For a CSSE-enabled platform protecting web applications, the instrumentation of the HTTP input vector is the most important, as this is the normal avenue for user-provided input. Other input vectors include application parameters (e.g., environment or run-time parameters) and data read from a persistent storage (e.g., databases and XML files).

If the application uses the persistent storage as both an input vector and an output vector, higher-order injections can occur. To prevent this, CSSE also requires that the metadata be made persistent so that is can be restored later. If the CSSE implementation does not support this functionality, it may not be able to prevent all higher-order injection attacks. In such a scenario, CSSE could mark all the input from persistent storage as untrusted, which would prevent higher-orders attacks but may result in false positives (cf. Section 6.2).

CSSE can also provide a programming interface to access the metadata directly. This allows the application developer to address special cases, such as when data read from a potentially unsafe source is explicitly trusted, or when untrusted data is used in a non-typical context.

Metadata-Preserving String Operations. As we have seen in Section 2, output expressions are typically constructed from application constants and user-provided input, using a chain of string operations, e.g., concatenation, substrings, case conversion or regexp matching.

We want to make sure that the metadata assigned to the string variables "survives" this chain of operations. Similar to the instrumentation of the input vectors for the metadata assignment, CSSE also instruments the string functions provided by the platform. These instrumented string functions are metadata-aware, and will update the metadata of their operands.

The complexity of the instrumentation depends on the particular string function. In many cases, this will be trivial, e.g., a function that changes the case of a string does not change the origin of the string fragments and thus only copying of the metadata is required. In other cases, more logic might be needed, e.g., string concatenation of two strings involves merging the metadata of the two strings. The number of string operations in a platform is typically quite large, and for CSSE to be complete, the entire set requires instrumentation.

The metadata in CSSE uses a string abstraction, as opposed to the lower-level internal representation of strings (i.e., byte or character arrays). In the rare cases where applications manipulate the internal representation of the data directly,CSSE might not be able to ensure up-to-date metadata. This can potentially lead to false positives or false negatives (cf. Section 6.2).

The three parts discussed above, form the metadata tracking system of CSSE. When these parts are implemented, it is possible to distinguish between the user-provided and developer-provided parts of the output expressions at any stage of their creation. This satisfies the first condition mentioned earlier.

Context-Sensitive String Evaluation. Context-sensitive string evaluation is the final part of CSSE, and is responsible for determining and executing the checks that ensure strict channel separation in the output expressions. This is again

achieved by an instrumentation of the platform, in this case the output vectors. This ensures that when the application calls an output vector to "execute" an output expression, CSSE is able to intercept the execution.

At this point, the complete context is known. The metadata of the output expression describes the origin of the data and thus determines parts of the expression that require checking. The function called by the application provides the second part of the context: the output vector the expression is intended for, and, following from this, the required checks.

For example, when an application calls `mysql_query()`, the CSSE instrumentation of this output vector intercepts this call. As CSSE instruments the function called, it is also aware that the function is responsible for the MySQL output vector and can thus determine the required checks on the untrusted fragments of the output expression.

For some output vectors, CSSE has to perform a limited syntactic analysis of the output expression. This is illustrated with the example of Section 2.1. In a single SQL query, the string constant and numerical constant have different interpretations and thus require different checks. Another example is HTML, where the same is true for elements, attributes and character-data parts. The complexity of the syntactic analysis required depends on the output vector.

When CSSE detects user-originated variable fragments that carry syntactic content in a given context, it is able to prevent the injection attack or raise an alert. The actual measures taken for preventing the attack depend on both the implementation and the particular output vector. Typically, CSSE escapes the offending content or blocks the request.

5 Implementation

CSSE is a generally applicable method, not tied to any particular platform. There are, however, several reasons why we chose PHP [14] as the target platform for our prototype implementation. First, PHP applications are particularly prone to injection vulnerabilities, owing to the lack of strict typing and proper APIs for data serialization. Second, numerous open-source PHP web applications are available, which allows us to easily validate our method and implementation. Finally, the platform itself is open-source, which enabled us to make the modifications described in this section.

CSSE can be implemented in different layers of the software stack. In particular, CSSE can be implemented either in the application itself or in the platform executing the application. The former requires modifications to the application, which need to be automated to retain one of the most important advantages of CSSE, namely, that it does not rely on the application developer. This can be achieved using a source code preprocessor that instruments the relevant function calls and operations. A more elegant and flexible solution makes use of the aspect-oriented programming [5] (AOP) paradigm to weave the necessary functionality into the application code, either at compile or at run time. As AOP implementations for PHP [18] do not yet support the necessary features (intercepting string

operations, not merely function calls), in our prototype we implemented CSSE using the second approach, i.e., by modifying the PHP platform.

The modifications to the PHP platform, comprised of the PHP interpreter and run-time libraries, entail the implementation of the four CSSE parts described in Section 4: metadata representation, metadata assignment, metadata-preserving string operations and context-sensitive string evaluation. Implementing these in an existing platform is not a trivial task and, in the case of PHP, involves numerous changes to a sparsely documented C code.

The goal of our prototype implementation of CSSE is threefold. First, it is a tool to illustrate our method and gain insight in the complexity involved in implementing it for an existing platform. Second, it allows us to test and demonstrate its effectiveness on a real-world application. Finally, it provides us with an estimate of the performance impact incurred by CSSE. As a goal of our prototype is a proof of concept, we have implemented the four parts of CSSE described in Section 4 up to the level that they support the aforementioned goal.

The prototype described here is based on the version 5.0.2 of the PHP platform. We modified it such that CSSE can be selectively turned on or off depending on the particular application being executed. The scope of our implementation is to prevent SQL injections in web applications. Therefore, for the input vectors, we focused on those related to HTTP, i.e., GET, POST, cookies and other HTTP parameters, and for the output vectors we focused on MySQL. Our prototype implements the four CSSE parts as follows:

Metadata Representation. CSSE requires that every string variable originating from user input have metadata associated with it. In our prototype we use a central metadata repository, which is implemented as a hash table indexed by the `zval` pointer — a dynamic memory structure representing a variable in PHP.

The metadata itself is represented as a bitmap of the length of a string, indicating the origin of each character. Currently, we use only one bit of information per character, to indicate whether certain data is user-provided. As discussed in Section 4, the remaining bits can be used to keep track of different possible origins of the data (e.g., user input, data read from the database, escaped user input and escaped data read from the database).

String variables that contain only parts that are not user-provided are identified by the absence of metadata. This improves both run-time performance and memory efficiency. It should, however, be noted that memory efficiency was not one of the design goals of our prototype implementation. By using more efficient memory representation, the memory efficiency of our prototype could be substantially improved.

Metadata Assignment. When an HTTP request is received by the PHP engine, all user input is imported into PHP variable space. We instrumented the appropriate functions to associate the proper metadata with each of the variables during the import phase. In addition, we also mark all strings read from the database as untrusted, thereby preventing second-order attacks (cf. Table 1).

Assigning metadata to variables imported from the environment and HTTP requests (GET, POST, cookies and authentication information) required modifi-

cations to only one function, namely the one responsible for registering of external variables as PHP variables (`php_register_variable_ex`). Other input vectors (e.g., database input) require modifications to appropriate external modules, e.g., `ext/mysql` in the case of MySQL.

Metadata-Preserving String Operations. Once the appropriate metadata is assigned to a string variable, it has to be preserved and updated during the entire lifetime of this variable. To meet this requirement, we instrumented a set of important string operations to make them metadata-preserving. This set includes the most common string operations used in the creation of expressions and consists of string concatenation, string interpolation (e.g., `"constant $var1 $var2"`), and the function that escapes metacharacters in the data (`addslashes`), and was sufficient for performing the evaluation detailed in the next section. We identified that in a complete implementation, 92 functions (out of the total of 3468 functions in PHP) would require instrumentation. Note that in most cases the instrumentations involves copying the entire or parts of metadata associated with the input string.

String operations are very common in applications, and thus special care has to be taken to minimize the performance impact of CSSE on this type of operations. In a typical application, most string operations will be performed on operands that contain no metadata, i.e., on variables that are not user-provided. We have addressed this by implementing the metadata-preserving string operations in such a way that the overhead is negligible in the absence of metadata (one hash table lookup for each operand to check whether metadata exists).

Context-Sensitive String Evaluation. In our prototype we focused on MySQL, a very common output vector for web applications. This required the instrumentation of all the functions responsible for MySQL query execution. When these functions are called, they will use the available metadata and knowledge about the output vector to perform the necessary checks on the executed expressions.

When the function that sends the MySQL query to the database is called, it is intercepted by CSSE. Prior to execution, CSSE checks whether there is any metadata associated with the SQL expression and if so it performs the necessary checks on the untrusted parts. In the case of MySQL, we require a limited syntactical analysis of the expression that distinguishes between string constants (e.g., `SELECT * from table where user='$username'`) and numerical constants (e.g., `SELECT * from table where id=$id`). Our method removes all unsafe characters (unescaped single quotes in the first case and all non-numeric characters in the second case) before sending the query to the database server.

6 Experimental Results

This section focuses on testing of the effectiveness and performance of CSSE on a real-world PHP application. It is important to note that our prototype was designed without analyzing the source code of this application. Instead, we determined the set of string operations and input and output vectors relevant

for our prototype based upon our knowledge of web applications in general. This provides some credibility that our method is valid and will achieve similar results with other applications.

For our experiments, we opted for the popular open-source bulletin-board application phpBB [15], based on the following three reasons. First, phpBB is widely used and thus results are relevant to a large user community. Second, it has a high degree of complexity and thus our validation shows that the prototype works effectively on non-trivial examples. Finally, phpBB has been know for injection vulnerabilities in its older versions [16]. In our experiments we used version 2.0.0 (dated April 04, 2002), in which several injection vulnerabilities have been identified.

6.1 Preventing Injection Attacks

At the time of writing, there were 12 SQL injection vulnerabilities pertaining to phpBB v2.0.x in the Bugtraq database [16]. We were able to successfully reproduce seven attacks exploiting these vulnerabilities (Bugtraq IDs: `6634`, `9122`, `9314`, `9942`, `9896`, `9883`, `10722`). The other five were either specific to versions later than 2.0.0 or we were not able to reproduce them.

For our experiments, we applied the exploits for these vulnerabilities with CSSE disabled and made sure that they succeed. Subsequently, we enabled CSSE and repeated the attacks. The initial prototype prevented six of these seven attacks, without adversely affecting the usability of the phpBB. The seventh attack (Bugtraq ID `6634`), was also prevented after we instrumented an additional string function, `implode`, used by phpBB.

Examination of the source code reveals that by applying syntactic checks for HTML and script evaluation, our prototype would also prevent known XSS and script-injection vulnerabilities in phpBB. To illustrate how CSSE works, we will show how it prevented one of the seven vulnerabilities — the vulnerability with Bugtraq ID `9112`. The vulnerability is caused by the following code in `search.php`:

The variable `$search_id` has all the quotes escaped, either by PHP interpreter (if the "MagicQuotes" option is enabled) or automatically by the script and therefore the quotes are not a problem here. The problem is that the variable is used in a numerical context, where the metacharacters are any non-numerical characters. The condition in the comparison in line 4 evaluates to true when a non-zero numerical prefix in the variable exists, not when the variable contains only a numerical value (what the developer probably meant). As a result of this

```
1 { code }
2 $search_id = (isset($HTTP_GET_VARS['search_id'])) ? $HTTP_GET_VARS[ '
      search_id'] : '';
3 { code }
4   if ( intval($search_id) )
5   {
6     $sql = "SELECT search_array FROM " . SEARCH_TABLE . " WHERE search_id =
          $search_id AND session_id = '". $userdata['session_id'] . "'";
7       if (!($result = $db->sql_query($sql)))
8         { code }
9 { code }
```

invalid comparison, the code is vulnerable to injection attacks. For example, providing the following value as a `$search_id` variable "1 or 1=1", executes the following query in the database:

```
SELECT search_array FROM table WHERE search_id = 1 or 1=1 AND session_id =
    XXX
```

When CSSE is enabled, metadata associated with variable `$sql` marks the fragment "`1 or 1=1`" as originating from the user. Before the actual query is executed (line 7), CSSE parses the content of the above SQL query and determines that user-originated data (marked in gray) appears in the *numerical context*. Therefore, it detects and removes the part of user-originated data that is not allowed to occur in this context (marked in black). The result is the same as if the variable had been casted to an integer using `intval($search_id)` function, but the entire process is completely transparent to the application developer.

6.2 False Positives and False Negatives

There are two types of errors related to intrusion detection and prevention methods, generally referred to as false positives and false negatives. In this context, false positive are events, in which legitimate actions are considered malicious and therefore blocked. Conversely, false negatives are events, in which malicious actions go undetected.

We have shown that CSSE is an effective method for defending against injection attacks, however, in some situations false positives or false negatives can appear. We identified the following three scenarios:

Incomplete implementations. A complete CSSE implementation requires that all relevant input vectors, string operations and output vectors are instrumented. For example, when a string operation that is not instrumented is used on partially untrusted data, the metadata attached to this data might be lost or outdated. This may result in false positives or false negatives. Note that the lack of metadata may implicity mean that the entire string is safe (a "fail-safe" mode of CSSE) or unsafe (a "fail-secure" mode of CSSE). It depends on the particular application and requirements which mode should implemented.

Defending against higher-order injections requires special attention. For CSSE to correctly address this class of injection vulnerabilities, metadata associated with persistent data has to be made persistent as well. If this functionality is not implemented, as is the case with our prototype, this might lead to either false positives or false negatives depending on the default policy of input retrieved from persistent storage.

Incorrect implementations. A second scenario, in which false positives or false negatives might occur, is the incorrect implementation of one of the parts that make up CSSE. The instrumentation of the output vectors is the most complex part, as this requires a limited syntactic analysis of the output expressions, and is therefore most prone to implementation errors. This might result in either false positives or false negatives.

For example, in our SQL implementation, we assumed that a user-supplied part might occur in a string or numeric constant. This works well with MySQL, but other databases may require more complicated checks or syntactic analysis. Another example is related to XSS attacks. Whereas preventing all HTML tags in a text is very simple, preventing only *unsafe* HTML tags requires a more complex analysis of the document (e.g., even a potentially safe `<b>` tag can have a `onMouseOver` attribute followed by a script, Bugtraq ID: `6248`).

It is worth stressing that CSSE needs to be implemented only once per platform, and can therefore be developed by specialists and be subject to stringent testing.

Invalid assumptions. A third scenario pertains to the assumptions made in CSSE. In rare situations where these assumptions do not hold, this might again lead to false positives or false negatives.

One important assumption on which CSSE is built, is that user-provided data does not carry syntactic content. In some special cases we do trust user-provided data and thus allow the syntactic content in this data. In a CSSE-enabled platform this will result in false positives. However, there are two methods for alleviating this problem: CSSE can be selectively enabled depending on the application and certain data can be explicitly marked as trusted using a provided API.

The second assumption is related to the string representation. CSSE operates on a string-abstraction representation of data. When an application performs direct manipulations on the lower-level representation, e.g., a character array, CSSE might not be able to update the metadata properly. In such a situation, to prevent false positives or false negatives, metadata should be manually updated by the application developer using a provided API.

6.3 Run-Time Measurements

We also analyzed the impact of CSSE on the performance of PHP. We performed five tests in which we measured the execution time:

T1-cgi: Requesting the webpage `phpBB2/viewforum.php?f=1`, containing the content of one forum. This operation involves several database reads and writes (including creating and storing a new session ID). PHP was running as a CGI application.

T1-mod: The same test as `T1-cgi`, except that PHP was running as an Apache2 module.

T2-cgi: Requesting the webpage `phpBB2/profile.php?mode=editprofile&sid=`, containing the content of one forum with a valid session ID. This test involved several database reads and complex output formatting with many string operations (creating a complex form with user-supplied data). PHP was running as a CGI application.

T2-mod: The same test as `T2-cgi`, except that PHP was running as an Apache2 module.

T3-CLI: This test was the standard PHP test (script `run-tests.php`) included in PHP source code. This test runs tests designed by the PHP-platform

Table 2. Run-time overhead evaluation: execution time for different tests. Errors shown are 95% confidence intervals with sample size 500 (20 for the last column).

Test Name	T1 (phpbb2 get)		T2 (phpbb2 get)		T3 (PHP tests)
Type	CGI	mod_apache	CGI	mod_apache	CLI
Unpatched	61.67 ± 0.23 ms	61.12 ± 0.28 ms	58.59 ± 0.07 ms	57.87 ± 0.07 ms	21.19 ± 0.06 s
CSSE disabled	62.22 ± 0.24 ms	62.85 ± 0.29 ms	58.85 ± 0.06 ms	59.41 ± 0.08 ms	21.28 ± 0.05 s
CSSE enabled	66.42 ± 0.29 ms	71.54 ± 0.37 ms	61.29 ± 0.07 ms	66.63 ± 0.09 ms	21.67 ± 0.07 s

developers to test the correctness of PHP[1]. Note that these tests do not involve a web-server and are usually not I/O intensive, therefore the expected impact of CSSE should be lower than with T1 and T2.

The results obtained are shown in Table 2. Tests `T1-cgi`, `T1-mod`, `T2-cgi`, `T2-mod` were executed 600 times of which the first 100 times were discarded to prevent caching artifacts. The timings were calculated by the Apache server. Because of the long run time, the last test was executed only 20 times. The table also shows 95% confidence intervals for each set of experiments. All measurements were done on a single machine with a Pentium M processor running at 1.7 GHz, 1 GB of RAM, running Linux 2.6.8. We tested PHP in the following three configurations:

Unpatched: Normal PHP source code, compiled with the standard options.
CSSE disabled: PHP was patched to include CSSE; however, CSSE was disabled by the run-time configuration option. The overhead is checking the state of a PHP global configuration flag in each of the modified methods.
CSSE enabled: PHP was patched to include CSSE, and CSSE was enabled.

Note that the tests produced identical output in all three configurations, varying only in execution time.

6.4 Run-Time Overhead

We observed that the total run-time overhead does not exceed 8% of the total run time if PHP runs as a CGI application and is surprisingly higher, namely, 17%, if PHP runs as an Apache2 module. This is shown in Figure 3, where black bars represent the execution time of an unpatched PHP, grey bars show the overhead with CSSE disabled, and light grey bars indicate the overhead of CSSE. As expected, the performance overhead for non-I/O intensive operations (the last test with a standalone PHP interpreter), is only around 2% of the total execution time.

[1] In our experiments, 5 out of 581 tests run by `run-tests.php` failed (not all the modules were compiled and many other tests were skipped). This was not related to CSSE, and we obtained the same result with the original PHP code.

It is important to stress that these numbers should be interpreted in the context of the goals set for our prototype in Section 5. As the prototype is limited to the most commonly used string operations, our measurements will underestimate the actual performance impact. However, this underestimation is very small as the calls of the instrumented string functions account for a preponderance of the total number of string function calls. Additionally, our prototype is not optimized for performance and, for example, using an alternative metadata representation as `zval` values would have a positive impact on performance.

Contrary to our expectations, CSSE overhead was more than 2.5 times higher when PHP was running as a module, rather than as a CGI application, even with a simple flag check to determine whether CSSE is enabled. This is most likely due to some threading issues, resulting in loading the entire run-time configuration data in each string operation, which can possibly be avoided with more careful prototype design.

Another interesting observation is that PHP running as an Apache2 module does not yield any significant performance increase in comparison with a CGI application. We attribute this to our experiment setup, in which the PHP interpreter was already cached in the memory and was running only a single task. During normal operation, Apache2 modules are noticeably faster than CGI.

To conclude, the overall performance overhead is application-dependent. Our tests suggest that it ranges from 2% for applications with few I/O operations to around 10% for typical web applications with PHP running with a webserver.

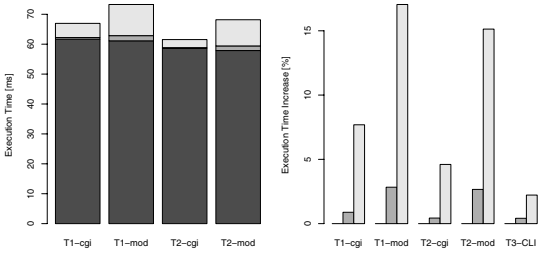

Fig. 3. Run-time overhead evaluation: request processing time and the relative increase for different tests. Black bars show total run time, gray bars show the run-time overhead with CSSE disabled and light gray bars show the overhead with CSSE enabled.

In the current implementation, strings containing at least one untrusted part consume twice as much memory as do their normal counterparts. To investigate the memory efficiency of CSSE we analyzed the heap allocation of CSSE-enabled PHP run with tests T1 and T2 using Valgrind [19]. In both cases the impact of CSSE was at around 2% (40kB increase for a total of ca. 2MB allocated heap). This is intuitive as only a small amount of memory allocated by PHP is used for storing PHP variables, only some of which contain strings with user data.

Obviously, these results are application-dependant, but should be similar for typical web applications.

As we already mentioned, various optimization techniques can be applied to reduce this additional memory storage, but this was beyond the scope of our prototype. Our results show that even with this inefficient implementation the memory impact is negligible.

7 Conclusions and Future Work

Injection vulnerabilities form an important problem in application-level security. In this work we identified the root cause of these vulnerabilities—the ad-hoc serialization of user-provided input. In addition, we provided a unifying view of injection vulnerabilities, which facilitates reasoning about this class of vulnerabilities and allows for the prediction of new types of related vulnerabilities.

Based on our improved understanding, we developed Context-Sensitive String Evaluation (CSSE), a novel method for defending against injection attacks. CSSE addresses the root cause of injection vulnerabilities by enforcing strict channel separation, while still allowing the convenient use of ad-hoc serialization. CSSE is transparent to the application developer, as the necessary checks are enforced at the platform level: neither modification nor analysis of the applications is required. As a result, it is advantageous over the two categories of related solutions: safe ad-hoc serialization and serialization APIs.

CSSE works by an automatic marking of all user-originated data with metadata about its origin and ensuring that this metadata is preserved and updated when operations are performed on the data. The metadata enables a CSSE-enabled platform to automatically carry out the necessary checks at a very late stage, namely when the output expressions are ready to be sent to the handling component. As at this point the complete context of the output expressions is known, CSSE is able to independently determine and execute the appropriate checks on the data it previously marked unsafe.

We developed a prototype implementation of CSSE for the PHP platform, and evaluated it with phpBB, a large real-life application. Our prototype prevented all known SQL injection attacks, with a performance impact of ca. 10%.

As ongoing work, we are instrumenting the remaining string operations and output vectors to prevent more sophisticated injection attacks, including XSS attacks, and evaluate CSSE with other applications. We will also develop an application-level implementation of CSSE for a platform that supports the aspect-oriented programming paradigm.

Acknowledgments

Many thanks to Andreas Wespi, Birgit Baum-Waidner, Klaus Julisch, James Riordan, Axel Tanner and Diego Zamboni of the Global Security Analysis Laboratory for the stimulating discussions and feedback. We also thank Frank Piessens of the Katholieke Universiteit Leuven for his valuable comments on this paper.

References

1. Anley, C.: Advanced SQL Injection In SQL Server Applications. Technical report, NGSSoftware Insight Security Research (2002).
2. Anley, C.: (more) Advanced SQL Injection. Technical report, NGSSoftware Insight Security Research (2002).
3. Boyd, S., Keromytis, A.: SQLrand: Preventing SQL injection attacks. In Jakobsson, M., Yung, M., Zhou, J., eds.: Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference. Volume 3089 of Lecture Notes in Computer Science., Springer-Verlag (2004) 292–304.
4. Descartes, A., Bunce, T.: Perl DBI. O'Reilly (2000).
5. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241 of Lecture Notes in Computer Science., Springer-Verlag (1997) 220–242.
6. Larson, E., Austin, T.: High coverage detection of input-related security faults. In: Proceedings of the 12th USENIX Security Symposium, Washington D.C., USENIX (2003) 121–136.
7. Lim, J.: ADOdb Database Abstraction Library for PHP (and Python). Web page at `http://adodb.sourceforge.net` (2000–2004).
8. Maor, O., Shulman, A.: SQL Injection Signatures Evasion. Technical report, Imperva Application Defense Center (2004).
9. Meijer, E., Schulte, W., Bierman, G.: Unifying tables, objects and documents. In: Workshop on Declarative Programming in the Context of OO Languages (DP-COOL'03), Uppsala, Sweeden (2003) 145–166.
10. MITRE: Common Vulnerabilites and Exposures. Web page at `http://cve.mitre.org` (1999–2004).
11. NIST: ICAT Metabase. Web page at `http://icat.nist.gov/` (2000–2004).
12. Ollmann, G.: HTML Code Injection and Cross-site Scripting. Technical report, Gunter Ollmann (2002).
13. Ollmann, G.: Second-order Code Injection Attacks. Technical report, NGSSoftware Insight Security Research (2004).
14. PHP Group, T.: PHP Hypertext Preprocessor. Web page at `http://www.php.net` (2001–2004).
15. phpBB Group, T.: phpBB.com. Web page at `http://www.phpbb.com` (2001–2004).
16. SecurityFocus: BugTraq. Web page at `http://www.securityfocus.com/bid` (1998–2004).
17. Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting format string vulnerabilities with type qualifiers. In: Proceedings of the 10th USENIX Security Symposium, Washington D.C., USENIX (2001) 257–272.
18. Stamey, J.W., Saunders, B.T., Cameron, M.: Aspect Oriented PHP (AOPHP). Web page at `http://www.aophp.net` (2004–2005).
19. Valgrind Developers: Valgrind. Web page at `http://valgrind.org` (2000–2005).
20. Wall, L., Christiansen, T., Orwant, J.: Programming Perl. O'Reilly (2000).