PHP Aspis: Using Partial Taint Tracking To Protect Against Injection Attacks

Ioannis Papagiannis
Imperial College London

Matteo Migliavacca
Imperial College London

Peter Pietzuch
Imperial College London

Abstract

Web applications are increasingly popular victims of security attacks. Injection attacks, such as Cross Site Scripting or SQL Injection, are a persistent problem. Even though developers are aware of them, the suggested best practices for protection are error prone: unless all user input is consistently filtered, any application may be vulnerable. When hosting web applications, administrators face a dilemma: they can only deploy applications that are trusted or they risk their system's security.

To prevent injection vulnerabilities, we introduce PHP Aspis: a source code transformation tool that applies *partial taint tracking* at the language level. PHP Aspis augments values with taint meta-data to track their origin in order to detect injection vulnerabilities. To improve performance, PHP Aspis carries out taint propagation only in an application's most vulnerable parts: third-party plugins. We evaluate PHP Aspis with Wordpress, a popular open source weblog platform, and show that it prevents all code injection exploits that were found in Wordpress plugins in 2010.

1 Introduction

The most common types of web application attacks involve code injection [4]: Javascript that is embedded into the generated HTML (Cross Site Scripting, or XSS), SQL that is part of a generated database query (SQL Injection, or SQLI) or scripts that are executed on the web server (Shell Injection and Eval Injection). These attacks commonly exploit the web application's trust in user-provided data. If user-provided data are not properly filtered and sanitised before use, an attacker can trick the application into generating arbitrary HMTL responses and SQL queries, or even execute user-supplied, malicious code.

Even though web developers are generally aware of code injection vulnerabilities, applications continue to suffer from relevant exploits. In 2010, 23.9% of the total reported vulnerabilities to the CVE database were classified as SQLI or XSS [12]. Morover, injection vulnerabilities are often common in third-party plugins instead of the well-tested core of a web application: in 2010, 10 out of 12 reported Wordpress injection exploits in the CVE database involved plugins and not Wordpress itself.

Such vulnerabilities still remain because suggested solutions often require manual tracking and filtering of user-generated data throughout the source code of an application. Yet, even a single unprotected input channel in an application is enough to cause an injection vulnerability. Thus, less experienced and therefore less security-conscious developers of third-party plugins are more likely to write vulnerable code.

Past research has suggested *runtime taint tracking* [19, 18, 14] as an effective solution to prevent injection exploits. In this approach, the origin of all data within the application is tracked by associating meta-data with strings. When an application executes a sensitive operation, such as outputting HTML, these meta-data are used to escape potentially dangerous values. The most efficient implementation of taint tracking is within the language runtime. Runtime taint tracking is not widely used in PHP, however, because it relies on custom runtimes that are not available in production environments. Thus, developers are forced to avoid vulnerabilities manually.

We show that injection vulnerabilities in PHP can be addressed by applying taint tracking entirely at the source code level without modifications to the PHP language runtime. To reduce the incurred performance overhead due to extensive source code rewriting, we introduce *partial taint tracking*, which limits taint tracking only to functions of the web application in which vulnerabilities are more likely to occur. Partial taint tracking effectively captures the different levels of trust placed into different parts of web applications. It offers better performance because parts of the application code remain unchanged.

We demonstrate this approach using PHP Aspis[1], a

[1] An Aspis was the circular wooden shield carried by soldiers in ancient Greece.

tool that performs taint tracking only on third-party plugins by rewriting their source code to explicitly track and propagate the origin of characters in strings. PHP Aspis augments values to include taint meta-data and rewrites PHP statements to operate in the presence of taint meta-data and propagate these correctly. PHP Aspis then uses the taint meta-data to automatically sanitise user-provided untrusted values and trasparently prevent injection attacks. Overall, PHP Aspis does not require modifications to the PHP language runtime or to the web server.

Our evaluation shows that, by using partial taint tracking, PHP Aspis successfully prevents most XSS and SQLI exploits reported in public Wordpress plugins since 2010. Page generation time is significantly reduced compared to tracking taint in the entire Wordpress codebase.

In summary, the contributions of this paper are:

- a taint tracking implementation for PHP that uses source code transformations only;
- a method for applying taint tracking only to parts of a web application, in which exploits are more likely to occur;
- an implementation of a code transformation tool and its evaluation with real-world exploits reported in the Wordpress platform.

The next section provides background on code injection vulnerabilities and introduces partial taint tracking as a suitable defence. In §3, we describe our approach for achieving taint propagation in PHP based on code rewriting, and we show how to limit the scope of taint tracking to parts of a codebase. Finally, we evaluate our approach by securing Wordpress in §4 and conclude in §5.

2 Preventing Injection Vulnerabilities

2.1 Injection Vulnerabilities

Consider a weblog with a search field. Typically, input to the search field results in a web request with the search term as a parameter:

```
http://goodsite.com/find?t=spaceship
```

A response of the web server to this request may contain the following fragment:

```
<p> The term ``spaceship'' was not found. </p>
```

The important element of the above response is that the user-submitted search term is included as is in the output. This can be easily exploited by an attacker to construct an XSS attack. The attacker first creates the following URL:

```
http://goodsite.com/find?t=<script\%20
src='http://attack.com/attack.js'/>
```

When an unsuspecting user clicks on this link, the following HTML fragment is generated:

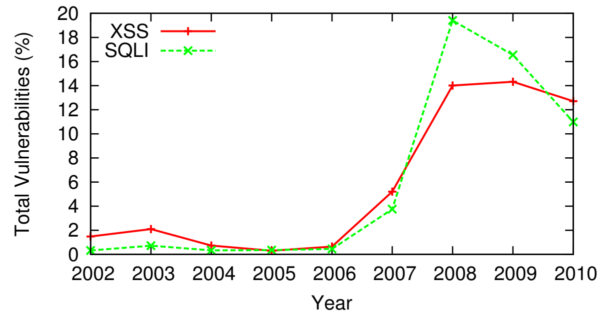

Figure 1: Historical percentage of XSS and SQLI in all CVE vulnerabilities

	Num. of Occurrences	
Type	*Wordpress*	*Drupal*
Cross Site Scripting	9	28
SQL Injection	3	1
Information Leakage	1	0
Insufficient Access Control	1	9
Eval Injection	0	1
Cross Site Request Forgery	1	0

Table 1: Web application vulnerabilities in 2010

```
<p> The term ``<script src='http://attack.com/
    attack.js' />'' was not found. </p>
```

The victim's browser then fetches the malicious Javascript code and executes it. Since the HTML document originated from `goodsite.com`, the script is executed with full access to the victim's web browser session. If the user has an account with this weblog and is already logged on, their cookies can be sent to the attacker. In general, the script can issue any unverified operations on behalf of the user.

SQL Injection attacks are analogous: they take advantage of applications that process user input to form an SQL query to a database. Similarly, Eval and Shell Injection attacks target PHP's `eval()` and `exec()` statements respectively. Since these statements execute arbitrary code at runtime, such an attack can easily compromise the host machine.

Figure 1 shows the percentage of reported SQLI and XSS vulnerabilities in recent years, as classified in the CVE database. Both problems continue to affect applications despite an increase of developer awareness. Table 1 shows our classification of vulnerabilities for two popular open source web platforms, Wordpress and Drupal. Code injection vulnerabilities are the most common type in both, with higher percentages compared to traditional applications across all programming languages in Figure 1. By relying on prepared statements, an automated way to avoid SQLI (see §2.2), the Drupal platform has comparatively few such exploits.

2.2 Existing Practices

The traditional approach to address injection vulnerabilities is to treat all information that may potentially come from the user as untrusted—it should not be used for sensitive operations, such as page generation or SQL queries, unless properly sanitised or filtered. In the previous example, a proper *sanitisation function* would translate all characters with special meaning in HTML to their display equivalents (e.g. "<" to "`<`"). Sanitisation functions such as `htmlentities()` or `escapeshellarg()` are part of the PHP language. After sanitisation, the string can safely be echoed to the client because it can no longer change the semantics of the output. SQLI filtering functions operate similarly but they also check for user-provided SQL keywords in the query [13, 19].

Unfortunately, sanitisation functions are difficult to apply in practice. Each sensitive operation requires a different sanitisation function. For example, if the same string is echoed to the user and used as part of an SQL query, two different strings must be generated based on the original value. Centralised filtering of input data when they are received is impractical because there is no single data representation that is both meaningful and secure in all possible contexts. For example, the string "`WHERE`" is safe in HTML but not in an SQL query. Therefore, developers have to propagate the original user data and only sanitise them before being used.

In addition, sanitisation assumes that developers can effectively track the origin of data and enforce that user-generated data always pass through their respective sanitisation function. In practice, left-out checks by inexperienced developers or unforeseen interactions that result in unexpected data flow (e.g. assuming that a script cannot be called from an external user) are likely to occur.

2.2.1 Static Approaches

Past research has suggested static analysis tools that detect injection vulnerabilities in PHP scripts. Pixy [11] and WebSSARI [10] rely on data flow analysis to detect sensitive functions that may receive user data without sanitisation and produce warnings. Wassermann and Su [16] model string values and operations as grammars and then inspect them before query operations to reduce the false positive rate for SQLI detection. Xie and Aiken [17] use symbolic execution to support PHP's dynamic features and report a low false positive rate.

Although static analysis tools do not introduce a runtime overhead, they are not fully automated, cannot support all PHP features and do not always achieve a low false positive rate. In addition, they cannot handle vulnerabilities that involve the file system or the database. As a result, such tools are not widely used for PHP development.

Prepared statements are a way to avoid SQLI exploits. Instead of concatenating queries, an application defines static placeholder queries with parameters filled in at runtime. Parameters passed to the placeholders cannot change the semantics of the query, as its structure is determined in advance when the statement is prepared. In practice, many PHP applications do not use them because they were not traditionally supported by PHP or MySQL and instead manually sanitise SQL queries.

2.2.2 Dynamic Taint Tracking

A dynamic approach to addressing injection vulnerabilities in existing applications when they occur is *runtime taint tracking* [19, 6]. It automates the tracking of the origin of data and enforces that data pass through their respective sanitisation functions. Runtime taint tracking involves three different steps:

1. **Data entry points.** All data entering the application that may originate from the user are transparently augmented with *taint meta-data.* The form of these meta-data may vary: from one bit that marks that a particular string is user-provided (or *tainted*) [14] to a pointer that links to arbitrary policy objects [19].
2. **Taint propagation.** As the application processes data, the runtime system transparently propagates the associated taint meta-data. For example, when a tainted string is concatenated with another string, the result must be marked as tainted.
3. **Guarded sinks.** Every operation that can be used in an injection vulnerability (e.g. `echo()` and `eval()`) is intercepted. The interceptor examines the corresponding taint meta-data and calls the relevant sanitisation function or aborts the operation.

Taint tracking has been shown to be effective in securing existing web applications [19, 18, 9, 14, 6]. Compared to static approaches, it does not require either debugging or refactoring an existing codebase. Perl and Ruby support it in some form (through Perl's taint mode and Ruby's safe levels) but not PHP. Taint tracking can be applied to PHP by modifying the core of the PHP runtime [19]. Typically, it has been implemented by augmenting the interpreter's `zval` struct with taint data. Simple approaches [13, 14] assign one bit of taint meta-data per string character and propagate that meta-data to sinks independently of the sanitisation efforts of the application.

Later systems stored more meta-data per character in order to provide more fine-grained guarantees. Nemesis [7] uses two taint bits to automatically infer authentication and enforce access control. Resin [19] uses a pointer to arbitrary policy objects that can be also used to prevent injection vulnerabilities.

Xu et al. [18] suggest that a taint tracking implementation in C can be used to compile the PHP runtime and transparently add taint tracking support. Their approach ignores sanitisation efforts of the hosted PHP application and therefore suffers from false positives. Also, it does not support different policies for different applications running in the same runtime, a common scenario for many PHP deployments.

However, unless taint tracking is considered part of PHP and is officially adopted, third party implementations are impractical. As the PHP manual puts it:

> *"modifications to the Zend[2] engine should be avoided. Changes here result in incompatibilities with the rest of the world, and hardly anyone will ever adapt to specially patched Zend engines. Modifications can't be detached from the main PHP sources and are overridden with the next update using the "official" source repositories. Therefore, this method is generally considered bad practice"*

In the past, taint tracking support has been suggested as a feature to the PHP community but it has not been adopted, partly because of fears that it may lead to a false sense of security [15].

2.3 Partial Taint Tracking

PHP is the most popular web development language, as indicated by web surveys [2], and its gentle learning curve often attracts less experienced developers. Inexperienced developers are more likely to extend web applications through third-party code in the form of *plugins*.

Such extensibility is frequently a popular feature for web applications but leads to a significant security threat from plugins. In 2009, the CVE database reported that the Wordpress platform suffered from 15 injection vulnerabilities, out of which 13 were introduced by third-party plugins and only 2 involved the core platform. In 2010, the breakdown was similar: 10 vulnerabilities were due to plugins and only 2 due to Wordpress itself.

As a result, not all application code is equally prone to injection vulnerabilities. For example, Wordpress spends much of its page generation time in initialisation code, setting up the platform before handling user requests. This involves time-consuming steps such as querying the database for installed plugins, setting them up, and generating static parts of the response involving theme-aware headers and footers.

Injection vulnerabilities, on the other hand, tend to appear in code that handles user-generated content: CVE-2010-4257, an SQLI vulnerability, involved a function that handles track-backs after a user published a post;

[2]Zend is the name of the official PHP scripting engine

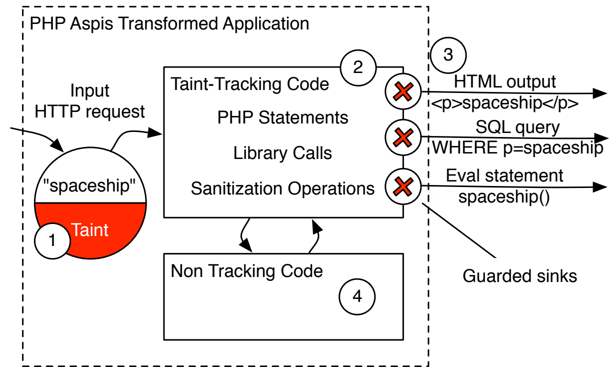

Figure 2: Partial taint tracking using PHP Aspis

CVE-2009-3891, an XSS vulnerability, involved a function that validates uploaded files; and CVE-2009-2851 and CVE-2010-4536, again XSS vulnerabilities, involved multiple functions that display user comments.

We exploit this observation by introducing *partial taint tracking*, which only transforms the source code of the most vulnerable parts of an application in order to support taint tracking. By relying on source-level transformations, partial taint tracking does not require the development and maintenance of a modified version of the PHP runtime.

We use a simple approach to decide when to track taint: we focus on parts of third-party plugins that handle user-generated data. This restricts source code transformations to a small fraction of the codebase of a web application. As a consequence, we mitigate the large performance penalty that exhaustive taint tracking at the source level would incur.

3 PHP Aspis

We describe the design and implementation of *PHP Aspis*, a PHP source code transformation tool for partial taint tracking. Figure 2 presents an overview of how PHP Aspis transforms applications. First, it modifies code that receives data from users and marks the data as user-generated (label 1). Second, it divides the application's codebase in two different categories: *tracking* and *non-tracking* code. Instead of tracking taint uniformly, it focuses on parts of the codebase that are more likely to contain code injection vulnerabilities (label 2). In tracking code, PHP Aspis records the origin of data at the character level and filters data at output statements when an injection vulnerability could exist (label 3). For non-tracking code, it does not perform taint tracking, trusting the code not to be vulnerable (label 4).

Next we introduce the representation of taint meta-data used to record the origin of the data in each variable. In §3.2, we describe the transformations that can be applied to PHP source code to (a) ensure its correct operation in

Sanitisation functions	`htmlentities()` `htmlspecialchars()`

Guarded sinks	`echo()` ⇒ `AspisAntiXss()`
	`print()` ⇒ `AspisAntiXss()`
	...

Table 2: Excerpt of the definition of the XSS taint category

the presence of taint meta-data; (b) propagate taint meta-data correctly; and (c) attach checks that inspect taint meta-data before each "sensitive" operation. Finally, we discuss how the untracked code can interact with the parts of the application that have been transformed to track taint in §3.3.

3.1 Taint Representation

PHP Aspis uses *character-level* taint tracking, i.e. tracks the taint of each string character individually [19]. Traditional variable-level taint tracking implementations (e.g. Ruby's safe levels) require the developer to explicitly untaint values before they are used. Instead, PHP Aspis prevents injection attacks transparently, and for this, it needs to know the exact characters that originate from the user. Consider for example an application that concatenates a user-provided value with a static HTML template, stores the result in `$v` and then returns `$v` to the client as a response. Inferring that the variable `$v` is tainted is of little use as `$v` also contains application-generated HTML. Instead, PHP Aspis uses character-level taint meta-data and only sanitises the parts of `$v` that originate from the user.

3.1.1 Taint Categories

PHP Aspis can track multiple independent and user provided *taint categories*. A taint category is a generic way of defining how an application is supposed to sanitise data and how PHP Aspis should enforce that the application always sanitises data before they are used.

Each taint category is defined as a set of *sanitisation functions* and a set of *guarded sinks*. Sanitisation functions can be PHP library functions or can be defined by the application. A sanitisation function is called by the application to transform untrusted user data so that they cannot be used for a particular type of injection attack. Commonly, sanitisation functions either transform unsafe character sequences to safe equivalents (e.g. `htmlentities`) or filter out a subset of potentially dangerous occurrences (e.g. remove `<script>` but not `<b>`). Calls to sanitisation functions by the application are intercepted and PHP Aspis untaints the corresponding data to avoid sanitising them again.

Guarded sinks are functions that protect data flow to sensitive sink functions. When a call to a sink function is made, PHP Aspis invokes the guard with references to the parameters passed to the sink function. The guard is a user-provided function that has access to the relevant taint category meta-data and typically invokes one or more sanitisation functions for that taint category.

For example, Table 2 shows an excerpt of an XSS taint category definition. It specifies that a user-provided string can be safely echoed to the user after either `htmlentities` or `htmlspecialchars` has been invoked on it. The second part exhaustively lists all functions that can output strings to the user (e.g. `echo`, `print`, etc.) and guards them with an external filtering function (`AspisAntiXss`). The guard either aborts the print operation or sanitises any remaining characters. The administrator can change the definitions of taint categories according to the requirements of the application.

By listing all the sanitisation functions of an application in the relevant taint category, PHP Aspis can closely monitor the application's sanitisation efforts. When applied to a well designed application, PHP Aspis untaints user data as they get sanitised by the application, before they actually reach the sink guards. Thus, sink guards can apply a simple, application agnostic, sanitisation operation (e.g. `htmlentities`) acting as a "safety net".

On the other hand, an application may not define explicit sanitisation functions or these functions may be omitted from the relevant taint category. In such cases, sink guards have to replicate the filtering logic of the application. In general, however, sink guards lack contextual information and this prevents them from enforcing context-aware filtering, e.g. guards cannot enforce sanitisation that varies according to the current user.

A different taint category must be used for each type of injection vulnerability. PHP Aspis tracks different taint categories independently from each other. For example, when a sanitisation function of an XSS taint category is called on a string, the string is still considered unsanitised for all other taint categories. This ensures that a sanitisation function for handling one type of injection vulnerability is not used to sanitise data for another type.

3.1.2 Storing taint meta-data

It is challenging to represent taint meta-data so that it supports arbitrary taint categories and character-level taint tracking. This is due to the following properties of the PHP language:

- *P1* PHP is not object-oriented. Although it supports objects, built-in types such as `string` cannot be augmented transparently with taint meta-data. This precludes solutions that rely on altered class libraries [6].
- *P2* PHP does not offer direct access to memory. Any solution must track PHP references because variables' memory addresses cannot be used [18].

String	Taint meta-data
`$s='Hello'`	`array(0=>false)`
`$n='John'`	`array(0=>true)`
`$r=$s.$n`	`array(0=>false, 5=>true)`

Table 3: Representation of taint meta-data for a single taint category

P3 PHP uses different assignment semantics for objects ("by reference") compared to other types including arrays ("by copy"). This does not allow for the substitution of any scalar type with an object without manually copying objects to avoid aliasing.

P4 PHP is a dynamically typed language, which means that there is no generic method to statically identify all string variables.

Due to these properties, our solution relies on PHP arrays to store taint meta-data by enclosing the original values. Table 3 shows how PHP Aspis encodes taint meta-data for a single taint category. For each string, PHP Aspis keeps an array of character taints, with each index representing the first character that has this taint. In the example, string `$s` is untainted, `$n` is tainted and their concatenation, `$r`, it untainted from index 0 to 4, and tainted from index 5 onwards. Numerical values use the same structure for taint representation but only store a common taint for all digits.

Taint meta-data must remain associated with the value that they refer to. As shown in Table 4, we choose to store them together. First, all scalars such as `'Hello'` and `12` are replaced with arrays (rows 1 and 2). We refer to this enclosing array as the value's *Aspis*. The Aspis contains the original value and an array of the taint meta-data for all currently tracked taint categories (`TaintCats`). Similarly, scalars within arrays are transformed into Aspis-protected values.

According to P4, PHP lacks static variable type information. Moreover, it offers type identification functions at runtime. When scalars are replaced with arrays, the system must be able to distinguish between an Aspis-protected value and a proper array. For this, we enclose the resulting arrays themselves in an Aspis-protected value, albeit without any taint (`false` in rows 3 and 4). The original value of a variable can always be found at index 0 when Aspis-protected. Objects are handled similarly. Aspis-protected values can replace original values in all places except for array keys: PHP arrays can only use the types `string` or `int` as keys. To circumvent this, the key's taint categories are attached to the content's Aspis (`KeyTaintCats`) and the key retains its original type (row 5).

Overall, this taint representation is compatible with the language properties mentioned above. By avoiding storing taint inside objects, we ensure that an assignment cannot lead to two separate values referencing the same taint category instances (P3). By storing taint categories in place, we ensure that variable aliasing correctly aliases taints (P2). Finally, by not storing taint meta-data separately, the code transformations that enable taint propagation can be limited to correctly handling the original, Aspis-protected values. As a result, the structure of the application in terms of functions and classes remains unchanged, which simplifies interoperability with non-tracking code as explained in § 3.3.

	Original value	Aspis-protected value

1.	**`'Hello'`**	`array(`
		`'Hello',TaintCats`
		`)`

2.	**`12`**	`array(12,TaintCats)`

3.	**`array()`**	`array(`
		`array()`
		`,false)`

4.	**`array('John')`**	`array(`
		`array(`
		`array(`
		`'John',TaintCats`
		`)`
		`)`
		`,false)`

5.	**`array(13=>20)`**	`array(`
		`array(13=>`
		`array(`
		`20,ContentTaintCats,`
		`KeyTaintCats`
		`)`
		`)`
		`,false)`

Table 4: Augmenting values with taint meta-data

3.2 Taint-tracking Transformations

Based on this taint representation, PHP Aspis modifies an application to support taint tracking. We refer to these source code transformations as *taint-tracking transformations*. These transformations achieve the three steps described in §2.2.2 required for runtime taint tracking: data entry points, taint propagation and guarded sinks.

3.2.1 Data Entry Points

Taint-tracking transformations must mark any user-generated data as fully tainted, i.e. all taint meta-data for every taint category in an Aspis-protected value should be set to `true`. Any input channel such as the incoming HTTP request that is not under the direct control of the application may potentially contain user-generated data.

Original expression	Transformed expression
`$s.$t`	`concat($s,$t)`
`$l = &$m`	`$l=&$m`
`$j = $i++`	`$j=postincr($i)`
`$b+$c`	`add($b,$c)`
`if ($v) {}`	`if ($v[0]) {}`
`foreach ($a as $k=>$v) {...}`	`foreach ($a[0] as $k=>$v) {restoreTaint($k,$v)...}`

Table 5: Transformations to propagate taint and restore the original semantics when Aspis-protected values are used

In each transformed PHP script, PHP Aspis inserts initialisation code that (1) scans the superglobal arrays to identify the HTTP request data, (2) replaces all submitted values with their Aspis-enclosed counterparts and (3) marks user submitted values as fully tainted. All constants defined within the script are also Aspis-protected, however, they are marked as fully untainted (i.e. all taint meta-data for every taint category have the value `false`). As a result, all initial values are Aspis-protected in the transformed script, tainted or not.

3.2.2 Taint Propagation

Next all statements and expressions are transformed to (1) operate with Aspis-protected values, (2) propagate their taint correctly and (3) return Aspis-protected values.

Table 5 lists some representative transformations for common operations supported by PHP Aspis. Functions in the right column are introduced to maintain the original semantics and/or propagate taint. For example, `concat` replaces operations for string concatenating in PHP (e.g. double quotes or the concat operator ".") and returns an Aspis-protected result. Control statements are transformed to access the enclosed original values directly. Only the `foreach` statement requires an extra call to `restoreTaint` to restore the taint meta-data of the key for subsequent statements in the loop body. The meta-data is stored with the content in `KeyTaintCats`, as shown in row 5 of Table 4.

PHP function library. Without modification, built-in PHP functions cannot operate on Aspis-protected values and therefore do not propagate taint meta-data. Since these functions are commonly compiled for performance, PHP Aspis uses *interceptor functions* to intercept calls to them and attach wrappers for taint propagation.

By default, PHP Aspis uses a generic interceptor for built-in functions. The generic interceptor reverts Aspis-protected parameters to their original values and wraps return values to be Aspis-protected again. This default behaviour is acceptable for library functions that do not propagate taint based on their semantics (e.g. `fclose`). However, the removal of taint from result values may lead to false negatives in taint tracking. PHP Aspis therefore provides custom interceptor functions for specific built-in functions. By increasing the number of intercepted functions, we improve the accuracy of taint tracking and reduce false negatives.

The purpose of a custom interceptor is to propagate taint from the input parameters to the return values. Such interceptors rely on the well defined semantics of the library functions to correctly propagate taint. When possible, the interceptor calculates the taint of the return value only based on the taints of the inputs (e.g. `substr`). It then removes the taint meta-data from the input values, invokes the original library function and attaches the calculated taint to the result value. Alternatively, the interceptor compares the result value to the passed parameter and infers the taint of the result. As an example, the interceptor for `stripslashes` compares the original and the result string and calculates the result's taint according to the slashes that are actually stripped from the original string. In total, 66 provided interceptors use this method.

For other functions, the interceptor can use the original function to automatically obtain a result with the correct taint. For example, `usort` sorts an array according to a user-provided callback function and thus can sort Aspis-protected values without changes. If the callback is a library function, the function is unable to compare Aspis-protected elements and calls to `usort` would fail. When callbacks are used, custom interceptors introduce a new callback replacing the old. That new callback calls the original callback after removing taint from its parameters. In total, 21 provided interceptors used this method.

In cases in which the result taint cannot be determined, such as for `sort`, an interceptor provides a separate, taint-aware version of the original PHP library function. We had to re-implement 19 library functions in this way.

Overall, our prototype currently intercepts 106 library functions. These functions include most of the standard PHP string library that, as we show in §4, are enough to effectively propagate taint in Wordpress.

Dynamic features. PHP has many dynamic features such as variable variables, variable function calls and the `eval` and `create_function` functions. These are not compatible with Aspis-protected values, but PHP Aspis must nevertheless maintain their correct semantics.

Variable variables only require access to the enclosed string. A dynamic access to a variable named `$v` is transformed from `$$v` to `${$v[0]}`. *Variable function calls* that use variables or library functions (e.g. `call_user_func_array`) allow a script to call a function that is statically unknown. PHP Aspis transforms these calls and inspects them at runtime. When a library

call is detected, PHP Aspis generates an interceptor, as described in the previous section, at runtime.

The functions `eval` and `create_function` are used to execute code generated at runtime. Since application generated code does not propagate taint, PHP Aspis must apply the taint-tracking transformations at runtime. To avoid a high runtime overhead, PHP Aspis uses a caching mechanism such as Zend Cache when available.

3.2.3 Guarded Sinks

PHP Aspis can protect code from injection vulnerabilities using the taint meta-data and defined taint categories. As described in §3.1.1, guard functions specified as part of active taint categories are executed before the calls to their respective sensitive sink functions. Guards use PHP's sanitisation routines (e.g. `htmlentities`) or define their own operations. For example, we use an SQL filtering routine that rejects queries with user-provided SQL operators or keywords [13].

3.3 Partial Taint Tracking

The taint tracking transformations used by PHP Aspis require extensive changes to the source code, which has an adverse impact on execution performance. To preserve program semantics, transformations often involve the replacement of efficient low-level operations by slower, high-level ones (see Table 5).

Partial taint tracking aims to improve execution performance by limiting taint tracking to the parts of the application in which injection vulnerabilities are more likely to exist. Partial taint tracking can be applied at the granularity of *contexts*: functions, classes or the global scope. The administrator can assign each of these to be of the following *types*: *tracking* or *non-tracking*.

Next we discuss how the presence of non-tracking code reduces the ability of PHP Aspis to prevent exploits. We also present the additional transformations that are done by PHP Aspis to support partial taint tracking.

3.3.1 Missed Vulnerabilities

When partial taint tracking is used, all code must be classified into tracking or non-tracking code. This decision is based on the trust that the application administrator has in the developers of a given part of the codebase. When parts of the codebase are classified as non-tracking, injection vulnerabilities within this code cannot be detected. On the other hand, PHP Aspis must still be able to detect vulnerabilities in tracking code. However, in the presence of non-tracking code, tracking code may not be the place where an exploit manifests itself and thus can be detected.

For example, a non-tracking function `n` in Figure 3 calls a tracking function `t` (step 1). It then receives a user-provided value `$v` from function `t` (step 2) and prints this

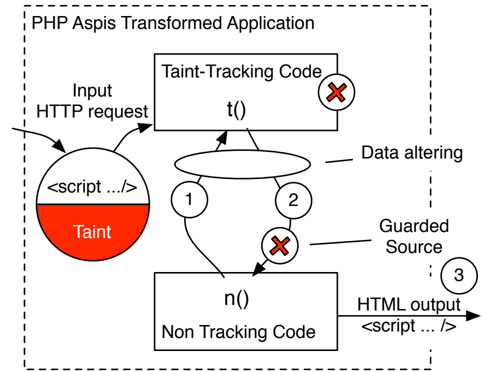

Figure 3: XSS vulnerability manifesting in non-tracking code

value (step 3). If `t` fails to escape user input, `n` is unable to sanitise the data transparently. This is because, in non-tracking code, taint meta-data are not available and calls to sensitive sinks such as `print` are not intercepted. From the perspective of `n`, `t` acts as the source of user generated data that must be sanitised before they leave the tracking context.

To address this issue, PHP Aspis takes a conservative approach. It can sanitise data at the boundary between tracking and non-tracking code. PHP Aspis adds *source guards* to each taint category to provide sanitisation functions for this purpose. A source is a tracking function and the guard is the sanitisation function applied to its return value when called from non-tracking code. In the above example, the tracking function `t` can act as a source of user generated data when called from `n`. A guard for `t` would intercept `t`'s return value and apply `htmlentities` to any user-generated characters. Source guards ensure that user data are properly sanitised before they can be used in non-tracking code.

Note though that this early sanitisation is an additional operation introduced by PHP Aspis. Thus, if the non-tracking context that received the data attempts to sanitise them again, the application would fail. Moreover, there is no generic sanitisation routine that can always be applied because the final use of the data is unknown. Instead, this solution is only suitable for cases when both the final use of the data is known and the application does not perform any additional sanitisation. This is often the case for third-party plugin APIs.

3.3.2 Compatibility Transformations

The taint-tracking transformations in §3.2 generate code that handles Aspis-protected values. For example, a tracking function that changes the case of a string parameter `$p` expects to find the actual string in `$p[0]`. Such a function can no longer be called directly from non-tracking code with a simple string for its parameter. Instead, PHP Aspis requires additional transformations to intercept this call and automatically convert `$p`

to an Aspis-protected value, which is marked as fully untainted. We refer to these additional transformations for partial taint tracking as *compatibility transformations*.

Compatibility transformations make changes to both tracking and non-tracking code. These changes alter the data that are exchanged between a tracking context and a non-tracking context, i.e. data exchanged between functions, classes and code in the global scope. They strip Aspis-protected values when passed to non-tracking contexts and restore Aspis protection for tracking contexts.

Function calls. A function call is the most common way of passing data across contexts. PHP Aspis transforms all cross-context function calls: a call from a tracking to a non-tracking context has its taint removed from parameters and the return value Aspis-protected again. The opposite happens for calls from non-tracking to tracking contexts. This also applies to method calls.

Adapting parameters and return values is similar to using the default interceptor function from §3.2. User code, however, can share objects of user-defined classes. Instead of adapting every internal object property, PHP Aspis uses proxy objects that decorate passed values. Consider an object `$o` of class `c` and assume that `c` is a tracking context. When `$o` is passed to the non-tracking context of function `f`, `f` is unable to access `$o`'s state directly or call its methods. Instead, it receives the decorator `$do` that points to `$o` internally. `$do` is then responsible for adapting the parameters and the return values of method calls when such calls occur. It also handles reads and writes of public object properties.

PHP also supports call-by-reference semantics for function parameters. Since changes to reference parameters by the callee are visible to the caller, these parameters effectively resemble return values. Compatibility transformations handle reference parameters similarly to return values—they are adapted to the calling context after the function call returns.

This behaviour can lead to problems if references to a single variable are stored in contexts of different types, i.e. if a tracking class internally has a reference to a variable also stored in a non-tracking class. In such cases, PHP Aspis can no longer track these variables effectively across contexts, forcing the administrator to mark both contexts as tracking or non-tracking. Since shared references to internal state make it hard to maintain class invariants, they are considered bad practice [5] and a manual audit did not reveal any occurrences in Wordpress.

Accessing global variables. PHP functions can access references to variables in the global scope using the `global` keyword. These variables may be Aspis-protected or not, dependent on the type of the current global context and previous function calls. The compatibility transformations rewrite `global` statements: when the imported variable does not match the context of the function, the variable is altered so that it can be used by the function. After the function returns, all imported global variables must be reverted to their previous forms—`return` statements are preceded with the necessary reverse transformations. When functions do not return values, reverse transformations are added as the last function statement.

Accessing superglobal variables. PHP also supports the notion of *superglobals*: arrays that include the HTTP request data and can be accessed from any scope without a `global` declaration. Data in these arrays are always kept tainted; removing their taint would effectively stop taint tracking everywhere in the application. As a result, only tracking contexts should directly access superglobals. In addition, compatibility transformations enable limited access from non-tracking contexts when access can be statically detected (i.e. a direct read to `$_GET` but not an indirect access through an aliasing variable). This is because PHP Aspis does not perform static alias analysis to detect such indirect accesses [11].

Include statements. PHP's global scope includes code outside of function and class definitions and spans across all included scripts. Compatibility transformations can handle different context types for different scripts. This introduces a problem for variables in the global scope: they are Aspis-protected when they are created by a tracking context but have their original value when they are created by a non-tracking context.

To address this issue, PHP Aspis alters temporarily all variables in the global scope to be compatible with the current context of an included script, before an `include` statement is executed. After the `include`, all global variables are altered again to match the previous context type. To mitigate the performance overhead of this, global scope code placed in different files but used to handle the same request should be in the same context type.

Dynamic features. Compatibility transformations intercept calls to `create_function` and `eval` at runtime. PHP Aspis then rewrites the provided code according to the context type of the caller: when non-tracking code calls `eval`, only the compatibility transformations are applied and non-tracking code is generated. Moreover, `create_function` uses a global array to store the context type of the resulting function. This information is then used to adapt the function's parameters and return value in subsequent calls.

3.4 Discussion

The taint tracking carried out by PHP Aspis is not precise. PHP Aspis follows a design philosophy that avoids uncertain taint prediction (e.g. in library functions without interceptors), which may result in false positives and affect application semantics. Instead, it favours a reduced ability to capture taint on certain execution paths, leading to false negatives. For this, it should not be trusted as the sole mechanism for protection against injection attacks.

Partial taint tracking is suited for applications where a partition between trusted and untrusted components is justified, e.g. third-party code. In addition, interactions across such components must be limited because if data flow from a tracking to non-tracking context and back, taint meta-data may be lost. PHP Aspis also does not track taint in file systems or databases, although techniques for this have been proposed in the past [8, 19].

PHP is a language without formal semantics. Available documentation is imprecise regarding certain features (e.g. increment operators and their side effects) and there are behavioural changes between interpreter versions (e.g. runtime call-by-reference semantics). Although our approach requires changes when the language semantics change, we believe that this cost is smaller than the maintenance of third-party runtime implementations that require updates even with maintenance releases.

Our taint tracking transformations support most common PHP features, as they are specified in the online manual [1]. We have yet to add support for newer features from PHP5 such as namespaces or closures.

4 Evaluation

The goals of our evaluation are to measure the effectiveness of our approach in preventing real-world vulnerabilities and to explore the performance penalty for the transformed application. To achieve this, we use PHP Aspis to secure an installation of *Wordpress* [3], a popular open source web logging platform, with known vulnerabilities.

We first describe how an administrator sets up PHP Aspis to protect a Wordpress installation. We then discuss the vulnerabilities observed and show how PHP Aspis addresses them. Finally, we measure the performance penalty incurred by PHP Aspis for multiple applications.

4.1 Securing Wordpress

Wordpress' extensibility relies on a set of hooks defined at certain places during request handling. User-provided functions can attach to these hooks and multiple types are supported: *actions* are used by plugins to carry out operations in response to certain event (e.g. send an email when a new post is published), and *filters* allow a plugin to alter data before they are used by Wordpress (e.g. a post must receive special formatting before being displayed).

CVE	Type	Guarded Sources	Prevented
2010-4518	XSS	1	Yes
2010-2924	SQLI	2	Yes
2010-4630	XSS	0	Yes
2010-4747	XSS	1	Yes
2011-0740	XSS	1	Yes
2010-4637	XSS	2	Yes
2010-3977	XSS	5	Yes
2010-1186	XSS	15	Yes
2010-4402	XSS	6	Yes
2011-0641	XSS	2	Yes
2011-1047	SQLI	1	Yes
2010-4277	XSS	3	Yes
2011-0760	XSS	1	No
2011-0759	XSS	9	No
2010-0673	SQLI	–	–

Table 6: Wordpress plugins' injection vulnerabilities; reported in 2010 and in the first quarter of 2011.

A plugin contains a set of event handlers for specific actions and filters and their initialisation code. Plugin scripts can also be executed through direct HTTP requests. In such cases, plugin scripts execute outside of the main Wordpress page generation process.

We secure a plugin from injection vulnerabilities using PHP Aspis as follows: first, we list the functions, classes and scripts defined by the plugin and mark the relevant contexts as *tracking*; second, we automatically inspect the plugin for possible sensitive sinks, such as print statements and SQL queries. We then decide the taint categories to be used in order to avoid irrelevant tracking (i.e. avoid tracking taint for eval injection if no `eval` statements exist); third, we obtain a list of event handlers from the `add_filter` statements used by the plugin. We augment the taint category definitions with these handlers as guarded sources because filters' return values are subsequently used by Wordpress (§3.3); and fourth, we classify the plugin initialisation code as non-tracking as it is less likely to contain injection vulnerabilities (§2.3).

4.2 Security

Table 6 lists all injection vulnerabilities reported in Wordpress plugins since 2010. For each vulnerable plugin, we verify the vulnerability using the attack vector described in the CVE report. We then try the same attack vector on an installation protected by PHP Aspis.

The experiments are done on the latest vulnerable plugin versions, as mentioned on each CVE report, running on Wordpress 2.9.2. PHP Aspis manages to prevent most vulnerabilities, which can be summarised according to three different categories:

Direct request vulnerabilities. The most common type of vulnerability involves direct requests to plugin scripts. Many scripts do not perform any sanitisation for some parameters (2010-4518, 2010-2924, 2010-4630, 2010-4747, 2011-0740). Others do not anticipate invalid parameter values and neglect to sanitise when printing error messages (2010-4637, 2010-3977, 2010-1186).

PHP Aspis manages to prevent all attack vectors described in the CVE reports by propagating taint correctly from the HTTP parameters, within the taint-transformed plugin scripts and to various printing or script termination statements such as such as `die` and `exit`, when its sanitisation functions are invoked.

Action vulnerabilities. Some of the plugins tested (2010-4402, 2011-0641, 2011-1047) introduce a vulnerability in an action event handler. Similarly, a few other plugins (2010-2924, 2010-1186, 2011-1047) only exhibit a vulnerability through a direct request but explicitly load Wordpress before servicing such a request. Wordpress transforms `$_GET` and `$_POST` by applying some preliminary functions to their values in `wp-settings.php`. As the Wordpress initialisation code is classified as non-tracking, it effectively removes all taint from the HTTP parameters and introduces a false negative for all plugins that execute after Wordpress has loaded.

Given that this behaviour is common for all plugins, we also use taint tracking in a limited set of Wordpress contexts—the functions `add_magic_quotes`, `esc_sql` and the `wpdb` class, which are invoked by this code. As the assignment statements that alter the superglobal tables are in the global scope of `wp-settings.php`, we also perform taint tracking in this context.

Unfortunately, this file is central in Wordpress initialisation: enabling taint tracking there leads to substantially reduced performance. To avoid this problem, we introduce a small function that encloses the assignment statements and we mark its context as tracking. This change required three extra lines of code to define and call the function but it significantly improved performance.

Filter vulnerabilities. From all tested plugins, only one (2010-4277) introduces a vulnerability in the code attached to a filter hook. Although we can verify the behaviour described in the CVE report, we believe that it is intended functionality of Wordpress: the Javascript injection is only done by a user who can normally post Javascript-enabled text. PHP Aspis correctly marks the post's text as untainted and avoided a false positive.

To test the filter, we edit the plugin to receive the text of posts from a tainted `$_GET` parameter instead of the Wordpress hook. After this change, PHP Aspis properly propagates taint and correctly escapes the dangerous Javascript in the guard applied to the filter hook.

4.2.1 False positives and negatives.

As discussed in §3.4, PHP Aspis may introduce both false negatives and false positives. By propagating taint correctly, we largely avoid the problem of false positives. False negatives, however, can be common because they are introduced (1) by built-in library functions that do not propagate taint, (2) by calls to non-tracking contexts and (3), by data paths that involve the file system or the database. In these cases, taint is removed from data, and when that data are subsequently used, vulnerabilities may not be prevented. PHP Aspis' current inability to track taint in the database is the reason why the XSS vulnerabilities 2011-0760 and 2011-0759 are not prevented.

To reduce the rate of false negatives, we use interceptors that perform precise taint tracking for all built-in library functions used by the tested plugins. In addition, we find that classifying the aforementioned set of Wordpress initialisation routines as tracking contexts is sufficient to prevent all other reported injection vulnerabilities. Note that the last vulnerable plugin (2010-0673) has been withdrawn and was not available for testing.

4.3 Performance

To evaluate the performance impact of PHP Aspis, we measure the page generation time for:

- a simple prime generator that tests each candidate number by dividing it with all smaller integers (Prime);
- a typical script that queries the local database and returns an HTML response (DB).
- Wordpress (WP) with the vulnerable Embedded Video plugin (2010-4277). Wordpress is configured to display a single post with a video link, which triggers the plugin on page generation.

Our measurements are taken in a 3 Ghz Intel Core 2 Duo E6850 machine with 4 GiB RAM, running Ubuntu 10.04 32-bit. We use PHP 5.3.3 and Zend Server 5.0.3 CE with Zend Optimizer and Zend Data Cache enabled. For each application, we enable tracking of two taint categories, XSS and SQLI.

Table 7 shows the 90th percentile of page generation times over 500 requests for various configurations. Overall, we observe that fully tracking taint has a performance impact that increases page generation between $3.4\times$ and $10.4\times$. The overhead of PHP Aspis depends on how CPU intensive the application is: DB is the least affected because its page generation is the result of a single database query. On the other hand, Prime has the worst performance penalty of $10.4\times$, mostly due to the replacement of efficient mathematical operators with function calls.

Wordpress (WP) with full taint tracking results in a $6.0\times$ increase of page generation time. With par-

App.	Tracking	Page generation	Penalty
Prime	Off	44.9 ms	-
Prime	On	466.8 ms	10.4×
DB	Off	0.4 ms	-
DB	On	1.3 ms	3.4×
WP	Off	65.6 ms	-
WP	On	394.4 ms	6.0×
WP	Partial	144.3 ms	2.2×

Table 7: Performance overhead of PHP Aspis in terms of page generation time

tial taint tracking configured only on the installed plugin, page generation overhead is significantly reduced to 2.2×. Given that Wordpress uses globals extensively, the main source of performance reduction for the partial taint tracking configuration are the checks on global variable access as part of the compatibility transformations.

Although full taint tracking at the source code level incurs a significant performance penalty, partial taint tracking can reduce the overhead considerably. In practice, 2.2× performance overhead when navigating Wordpress pages with partial taint tracking is acceptable for deployments in which security has priority over performance.

5 Conclusions

In this paper, we presented PHP Aspis, a tool that applies partial taint tracking at the source code level, with a focus on third-party extensions. PHP Aspis avoids the need for taint tracking support in the PHP runtime. Although the performance penalty of PHP Aspis can increase the page generation time by several times, we have shown that if taint tracking is limited only to a subset of a web application, the performance penalty is significantly reduced while many real world vulnerabilities are mitigated. Our evaluation with the Wordpress platform shows that PHP Aspis can offer increased protection when a moderate increase in page generation time is acceptable.

Acknowledgements. We would like to thank David Eyers and Evangelia Kalyvianaki for their comments on an earlier paper draft. This work was supported by the EP/F04246 grant from the UK Engineering and Physical Sciences Research Council.

References

[1] PHP online manual. www.php.net/manual.

[2] Programming Languages Popularity website. www.langpop.com.

[3] Wordpress website. www.wordpress.org.

[4] PHP Security Guide 1.0. Tech. rep., PHP Security Consortium, 2005.

[5] BLOCH, J. *Effective Java*, second ed. Prentice Hall, 2008.

[6] CHIN, E., AND WAGNER, D. Efficient character-level taint tracking for Java. In *Secure Web Services* (Chicago, IL, 2009), ACM.

[7] DALTON, M., KOZYRAKIS, C., AND ZELDOVICH, N. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *Security Symposium* (Montreal, Canada, 2009), USENIX.

[8] DAVIS, B., AND CHEN, H. DBTaint: Cross-Application Information Flow Tracking via Databases. In *WebApps* (Boston, MA, 2010), USENIX.

[9] HALFOND, W., ORSO, A., AND MANOLIOS, P. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering 34*, 1 (2008).

[10] HUANG, Y.-W., YU, F., ET AL. Securing web application code by static analysis and runtime protection. In *World Wide Web* (New York, NY, 2004), ACM.

[11] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Symposium on Security and Privacy* (Berkeley, CA, 2006), IEEE.

[12] MITRE CORPORATION. Common Vulnerabilities And Exposures (CVE) database, 2010.

[13] NGUYEN-TUONG, A., GUARNIERI, S., ET AL. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference* (Chiba, Japan, 2005).

[14] PIETRASZEK, T., AND BERGHE, C. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection* (Menlo Park, CA, 2006).

[15] VENEMA, W. Runtime taint support proposal. In *PHP Internals Mailing List* (2006).

[16] WASSERMANN, G., AND SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI* (San Diego, CA, 2007), ACM.

[17] XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *Security Symposium* (Vancouver, Canada, 2006), USENIX.

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

[19] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving application security with data flow assertions. In *SOSP* (New York, NY, 2009), ACM.