Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis

Philipp Vogt[§], Florian Nentwich[§], Nenad Jovanovic[§],
Engin Kirda[§], Christopher Kruegel[§], and Giovanni Vigna[‡]

[§] Secure Systems Lab
Technical University Vienna
{pvogt,fnentwich,enji,ek,chris}@seclab.tuwien.ac.at

[‡] University of California, Santa Barbara
vigna@cs.ucsb.edu

Abstract

Cross-site scripting (XSS) is an attack against web applications in which scripting code is injected into the output of an application that is then sent to a user's web browser. In the browser, this scripting code is executed and used to transfer sensitive data to a third party (i.e., the attacker). Currently, most approaches attempt to prevent XSS on the server side by inspecting and modifying the data that is exchanged between the web application and the user. Unfortunately, it is often the case that vulnerable applications are not fixed for a considerable amount of time, leaving the users vulnerable to attacks. The solution presented in this paper stops XSS attacks on the client side by tracking the flow of sensitive information inside the web browser. If sensitive information is about to be transferred to a third party, the user can decide if this should be permitted or not. As a result, the user has an additional protection layer when surfing the web, without solely depending on the security of the web application.

1 Introduction

Nowadays, many web sites make extensive use of client-side scripts (mostly written in JavaScript) to enhance the user's experience. Unfortunately, this trend has also increased the popularity and frequency of cross-site scripting (XSS) attacks. When a cross-site scripting vulnerability is present in a web application, an attacker can inject scripting code into the pages generated by the web application. This allows an attacker to circumvent the *same-origin policy* [19], which states that scripts loaded from a certain domain cannot access data belonging to any other domain.

In XSS attacks, this policy is circumvented because when the victim's browser receives the generated output page, the maliciously-injected code is executed in the context of the site hosting the vulnerable web application, and, therefore, it has access to sensitive data stored by that site in the victim's browser (e.g., using cookies). Usually, the attack code transfers the sensitive information to a server under the attacker's control. This information allows the attacker to impersonate the victim or hijack the victim's current session.

There are two general methods for injecting malicious code into the web page that is displayed to the user. In the first method, called *stored XSS*, the attacker persistently stores the malicious code in a resource managed by the web application, such as a database. The actual attack is carried out at a later time, when the victim requests a dynamic page that is constructed from the contents of this resource. As an example, consider a web-based bulletin board system (e.g., phpBB [25]) where people can post messages that are displayed to all visitors of the bulletin board. Let us assume further that the application does not remove script content from posted messages. In this case, the attacker can craft a message similar to the one in Figure 1. This message contains the malicious JavaScript code, which the bulletin board stores in its database. A visiting user who reads this message retrieves the scripting code as part of the message. The user's browser then executes the script, which, in turn, sends the user's cookie to a server under the attacker's control.

In the second method, called *reflected XSS*, the attack script is not persistently stored, but, instead, it is immedi-

```
Look at this picture!
<img src="image.jpg">
<script>
  document.images[0].src = "http://evilserver/image.jpg?
    stolencookie=" + document.cookie;
</script>
```

Figure 1. Transfer of a cookie.

ately "reflected" back to the user. For instance, consider a search form that includes the search query into the page with the results, but without filtering the query for scripting code. This vulnerability can be exploited, for example, by sending to the victim an email with a specially-crafted link that points to the search form and that contains the malicious JavaScript code. By tricking the victim into clicking this link, the search form is submitted with the JavaScript code as the query string, and the attack script is immediately sent back (reflected) to the victim, as part of the web page with the results.

The optimal approach to prevent XSS attacks would be to eliminate the vulnerabilities in the affected web applications. To this end, a web application must properly validate all input, and in particular, remove malicious scripts. The problem is that many service providers do not fix their web applications in a timely way, mostly due to financial constraints. Hence, a promising approach for protecting users against XSS attacks is to deploy the necessary security mechanisms on the client side.

The solution proposed in this paper uses dynamic data tainting. In contrast to traditional, tainting-based approaches on the server side, we taint sensitive information on the client side. The goal is to ensure that a JavaScript program can send *sensitive information* only to the site from which it was loaded. To this end, the information flow of sensitive data is tracked inside the JavaScript engine of the browser. Whenever an attempt to relay such information to a third party (i.e., the adversary) is detected, the user is warned and given the possibility to stop the transfer.

Unfortunately, it is not possible to detect all information flows dynamically [27]. This is a problem, because the adversary has complete freedom to craft his attack code, and, therefore, he could leverage information flows that are not covered by dynamic analysis to successfully launch XSS attacks. To address this limitation, we complement our dynamic mechanism with an additional static analysis component. This static analysis component is invoked on-demand and covers those cases that cannot be decided dynamically. Using a combination of static and dynamic analysis, we can combine the advantages of both approaches. The dynamic analysis allows us to precisely track sensitive information with low runtime overhead. By switching to static analysis when necessary, our system can provide stronger security in the face of malevolent attack code.

To demonstrate that our approach is capable of solving real-world problems, we integrated a prototype implementation of our techniques into the popular Firefox web browser (which turned out to be a considerable engineering effort). By further equipping Firefox with a web crawler capable of simulating user actions, we were able to conduct a large-scale and fully automatic evaluation of our system on more than one million web pages. The empirical results demonstrate that our approach provides reliable protection against XSS attacks in real-world usage, with a low false positive rate.
To summarize, the contributions of this paper are as follows:

- A dynamic taint analysis and a complementary static analysis that prevent XSS attacks by monitoring the flow of sensitive information in the web browser.
- The integration of the analyses into the popular Firefox web browser.
- The development of a Firefox-based web crawler capable of simulating user actions. This allowed us to perform a large-scale empirical validation of our techniques based on the automatic browsing of more than one million web pages.

The reminder of this paper is structured as follows. In Section 2 we present related work on detecting and preventing XSS attacks. In Section 3 we introduce our dynamic analysis technique. Then, in Section 4 we extend our approach using static techniques. Sections 5 and 6 discuss how information could be leaked to an adversary and some implementation issues, respectively. Section 7 presents the evaluation of our prototype, and, finally, Section 8 concludes.

2 Related Work

There are two main criteria that can be used for distinguishing between XSS protection techniques: The point of deployment (client-side or server-side), and the analysis paradigm in use (dynamic or static).

Server-side protection. A well-known, dynamic server-side protection mechanism is Perl's taint mode [3]. In this case, the flow of tainted values is tracked within the Perl interpreter. More precisely, input from untrusted sources is marked as being potentially malicious, and propagated through the program. Any attempt to use tainted data directly or indirectly in a critical command that invokes a subshell, modifies files, directories, or processes will be aborted with an error. The developer is given means to test the taint status of data, as well as the ability to sanitize (i.e., untaint) the data where this seems appropriate. Analogously,

interpreter-based approaches for PHP are presented in [23] and [26]. Dynamic taint propagation for the Java virtual machine is employed in [10].

A dynamic taint-tracking scheme for C programs utilizing source-to-source transformation is described in [30]. Here, the scope of protection ranges from classical buffer overflow and format string vulnerabilities to the detection of XSS and other types of injection attacks. Their method is also applicable to scripting languages implemented in C, such as PHP and Bash. In addition, there also exist dynamic tainting approaches that do not deal with XSS attacks, but focus on the detection of attacks that attempt to overwrite sensitive program data (such as return addresses or function pointers). For instance, in [22], binaries are rewritten at runtime to allow for taint propagation. Hardware approaches that dynamically track the propagation of taint values at the architectural level are presented in [6] and [28].

In [15], an anomaly-based intrusion detection system is presented that can detect XSS attacks. To this end, the system analyzes web server logs and automatically retrieves the profiles (length and structure) of typical parameters of any protected web application. These profiles are then compared to incoming user requests, such that requests with atypical parameter profiles can be classified as potential attacks.

Apart from the dynamic techniques mentioned so far, static analysis can be used to detect XSS vulnerabilities on the server side. In [16], the authors propose a static analysis approach for web applications in order to detect XSS vulnerabilities. The analysis results are then cross-checked with dynamic techniques to eliminate false warnings. A technique based on data flow analysis for detecting XSS and similar vulnerabilities (such as SQL injection or command injection) is presented in [13].

Client-side protection. There exist a few approaches that, similarly to our solution, try to solve the problem on the client side. In [12], the authors implemented a proxy that can be used to protect a user while surfing the web. To this end, the proxy analyzes the HTTP traffic exchanged between the user's browser and the contacted web server. First, client requests are scanned for special HTML characters (such as the `"<"` character). Then, if the application's response reflects these presumably-malicious request parameters back to the user, the web site is considered to be vulnerable to XSS. As a result, these special characters are encoded before the response is delivered to the user's browser, which disables the attempted attack. A limitation of this approach is that it is focused on reflected XSS attacks, and does not permit the detection and prevention of stored XSS attacks.

The application-level firewall described in [14] analyzes browsed HTML pages for hyperlinks that might lead to the leakage of sensitive data. Based on this analysis, a set of connection rules is generated on-the-fly that prevents suspicious requests. The main idea behind this technique is that sensitive information can be transmitted either by a single link that is constructed dynamically inside the user's browser, or by several static links.

In [11], the Mozilla web browser is equipped with an auditing system that monitors the execution of JavaScript code. Using different intrusion detection techniques, the observed operations are compared to high-level policies to detect malicious behavior.

The main difference between our solution and other client-based approaches is that they use various heuristics for XSS detection, whereas we perform an in-depth and precise analysis of how sensitive values are propagated inside the user's browser. Using a combination of dynamic and static analyses, we can efficiently identify implicit information flows that purely dynamic approaches cannot identify.

3 Dynamic Data Tainting

In a cross-site scripting attack, the code that is injected into the output of the web application is under the attacker's control. This code is executed in the user's web browser, where it collects sensitive information that is then sent to the attacker. Because the code runs in the context of the vulnerable web site, it is not distinguishable from normal application behavior.

The goal of our protection approach is to prevent that sensitive data is sent to a third party (the adversary) without the user's consent. To this end, we designed a mechanism that can keep track of how sensitive data is used in the browser. Our mechanism is based on the concept of *dynamic data tainting*, in which sensitive data is first marked (or tainted), and then, when this data is accessed by scripts running in the web browser, its use is dynamically tracked by our system. When tainted data is about to be transferred to a third party, different kinds of actions can be taken. Examples are logging, preventing the transfer, or stopping the program with an error. Note that it is sufficient to model the taint value associated with a piece of data as a simple boolean flag. In particular, it is not necessary to explicitly store the domain that this data originated from (to be able to distinguish the source domain from a third party domain), as this information can be retrieved from the browser.

Our taint analysis is capable of tracking data dependencies. That is, when a tainted value is assigned to another variable, this variable becomes tainted as well. Also, when any operand of an arithmetic or logic operation is tainted, the result becomes tainted. Moreover, our solution is capable of handling direct control dependencies. That is, whenever the execution of an operation depends on the value of a tainted variable (e.g., if an operation is guarded by an `if`-

Object	Tainted properties
Document	cookie, domain, forms, lastModified, links, referrer, title, URL
Form	action
Any form input element	checked, defaultChecked, defaultValue, name, selectedIndex, toString, value
History	current, next, previous, toString
Select option	defaultSelected, selected, text, value
Location and Link	hash, host, hostname, href, pathname, port, protocol, search, toString
Window	defaultStatus, status

Table 1. Initial sources of taint values.

```
 1: var cookie = document.cookie;
 2: // "cookie" is now tainted
 3: var dut = "";
 4: // copy cookie content to dut
 5: for (i=0; i < cookie.length; i++) {
 6:   switch (cookie[i]) {
 7:     case 'a': dut += 'a';break;
 8:     case 'b': dut += 'b';break;
 9:     ...
10:   }
11: }
12: // dut is now copy of cookie
13: document.images[0].src =
      "http://badsite/cookie?" + dut;
```

Figure 2. Attack using direct control dependency.

branch that tests a tainted variable), the result of this operation is tainted. Figure 2 provides an example to illustrate the importance of direct control dependencies. In this example, the attacker copies the cookie from the variable `cookie` to the variable `dut` using a for-loop and a switch statement for any character in `cookie`. If only data dependencies were covered, the `dut` variable would not be tainted after the loop. This is because the character literals assigned to it in the switch statement are not tainted. When direct control dependencies are considered, however, everything in the scope of the switch statement is tainted (because a tainted value is tested in the head of the switch statement).

In addition to the tracking of taint information inside the JavaScript engine, tainted data stored in and retrieved from the document object model (DOM) tree [29] of the HTML page has to retain its taint status. This is required to prevent laundering attempts in which an attacker temporarily stores tainted data in a DOM tree node to clear its taint status.

The next sections discuss the information flow in a typical script execution. We will first show what kind of information is considered sensitive in Section 3.1. Then, Section 3.2 presents how tainted data is propagated by our system when a script is run.

3.1 Sensitive Data Sources

For our system, we have to identify those data sources that are considered sensitive. The reason is that this data must be initially tainted so that its use by scripting code can be appropriately tracked. A data source is considered sensitive when it holds information that could be abused by an adversary to launch attacks or to learn information about a user (e.g., cookies or the URL of the visited web page). A list of tainted sources used by our system is provided in Table 1. Since this list was provided by Netscape [21], we believe it to be fairly complete. In case that additional sensitive data sources are discovered, our system can be easily extended to handle these as well.

Sensitive sources are directly tainted in the web browser. Thus, whenever a sensitive data element is accessed by a JavaScript program, we have to ensure that the result is regarded as tainted by the JavaScript engine as well. Figure 3 shows the interaction between the JavaScript engine and the browser when a script is executed. In this example, the HTML page contains some embedded JavaScript code (1) that accesses the cookie of the document (which is a sensitive source). The script is parsed and compiled into a bytecode program (2) that is then executed by the JavaScript engine. When the engine executes the statement that attempts to obtain the `cookie` property from the `document` object (3), it generates a call to the implementation of the document class in the browser (4). Possible parameters of the call are converted from values understood by the JavaScript engine to those defined in the browser (5). Then, the corresponding method in the browser, which implements the `document.cookie` method, is called (6). In this method, the access to a sensitive source is recognized and the value is tainted appropriately (7). This value is then converted into a value with a type used by the JavaScript engine (8). This conversion has to retain the taint status of the value. Thus, the result of the operation that obtains the `cookie` property (variable `x`) is tainted (9).

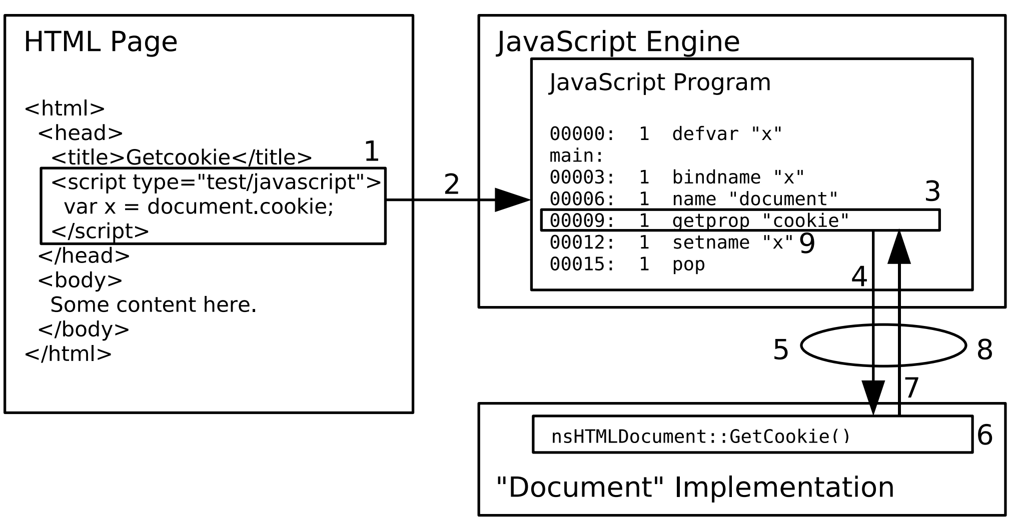

Figure 3. A JavaScript program accesses a sensitive source.

3.2 Taint Propagation

JavaScript programs that are part of a web page are parsed and compiled into an internal bytecode representation. These bytecode instructions are then interpreted by the JavaScript engine. To track the use of sensitive information by JavaScript programs, we have extended the JavaScript engine. More precisely, we have extended the semantics of the bytecode instructions so that taint information is correctly propagated. The JavaScript bytecode instructions can be divided into the following broad classes of operations:

- assignments;
- arithmetic and logic operations (`+`, `-`, `&`, etc.);
- control structures and loops (`if`, `while`, `switch`, `for in`);
- function calls and `eval`.

When an instruction is executed, some (or all) of its operands could be tainted. Thus, for each instruction, there has to be a rule that defines under which circumstances the result of an operation has to be tainted (or what other kind of information is affected by the tainted data).

3.2.1 Assignments

In an assignment operation, the value of the left-hand side is set. If the right-hand side of the assignment is tainted, then the target on the left-hand side is also tainted. The JavaScript engine has different instructions for assignment to single variables, function variables, function arguments, array elements, and object properties.

In some cases, the variable that is assigned a tainted value is not the only object that must be tainted. For example, if an element of an array is tainted, then the whole array object needs to be tainted as well. This is necessary to ensure that functions and methods that operate on the array as a whole, such as `arr.length`, return a tainted value. Consider the example in Figure 4. On Line 1, a new array is created with an initial length of 0. Only if the first character of the cookie is an '`a`', a value is assigned to the first element of the array on Line 3. In this example, the length of the array on Line 5 is 1 if the first character of the cookie is an '`a`', otherwise it is still 0. On Line 5, a new variable is set to '`a`', depending on the length of the array. When extending this method to cover all possible characters (e.g., '`a`' - '`z`', '`A`' - '`Z`', '`0`' - '`9`'), the attacker could try to copy the first character of the cookie to a new value, thereby attempting to bypass the tainting scheme. However, in our approach, we do not only taint the first element on Line 3, but also the array object itself. As a result, the variable `y` on Line 5 is tainted. Likewise, if a property of an object is set to a tainted value, then the whole object needs to be tainted. The reason is that the property could be new, and in this case, the number of properties has changed. This could allow an attacker to leak information in a similar fashion.

3.2.2 Arithmetic and Logic Operations

Operations in JavaScript can have one (e.g., unary -) or more operands (e.g., multiplication `*`). JavaScript, similar to Java bytecode, is a stack-based language. That is, in-

```
1: var arr = [ ];  // arr.length = 0
2: if (document.cookie[0] == 'a') {
3:   arr[0] = 1;
4: }
5: if (arr.length == 1) { y = 'a'; }
```

Figure 4. Array element assignment.

structions that perform arithmetic or logic operations first pop the appropriate number of operands from the stack and then push back the result. The result is tainted if one of the used operands is tainted.

3.2.3 Control Structures and Loops

Control structures and loops are used to manipulate the execution flow of a program and to repeat certain sequences of instructions (e.g., `if` constructs, `while` loops, and `try-catch-finally` blocks). If the condition of a control structure tests a tainted value, a *tainted scope* is generated that covers the whole control structure. The results of all operations and assignments in the scope are tainted. Note that introducing such a scope does *not* immediately taint all contained variables. Instead, a variable is dynamically tainted only when its value is modified inside a scope during the actual execution of the program. This is used to correctly handle direct control dependencies and prevent attempts of laundering tainted values by copying them to untainted values, as illustrated in Figure 2. In this example, a tainted value (`cookie.length`) is used in the termination condition of the `for`-loop on Line 4. Thus, a scope from Line 4 to Line 10 is generated. An additional scope is generated from Line 5 to 9, because the `switch`-condition is tainted. When processing operations within a tainted scope, the results of all operations are tainted, regardless of the taint status of any involved operands. Therefore, appending a character to the `dut` variable (e.g., on Line 6 in Figure 2) taints the `dut` variable. Note that this would not be the case if only data dependencies were considered.

`If-else` statements generate scopes for both branches when the condition is tainted. In `do-while` loops, the scope is not generated until the tainted condition is tested. As a result, the first time the block is executed, no scope is generated. If the tested condition is tainted, a new tainted scope covering the repeated block is generated, which remains until the loop is left. In the `try-catch-finally` statement, a scope is generated for the `catch`-block when the thrown exception object is tainted. The remaining control flow statements are handled analogously.

3.2.4 Function Calls and `eval`

Functions are tainted if they are defined in a tainted scope. For example, function `x` defined on Line 2 of Figure 5 is tainted, since a tainted scope has been created from Line 1

```
 1: if (document.cookie[0] == 'a') {
 2:   x = function () { return 'a'; };
 3:   // x is a tainted function
 4: }
 5: function func (par) { return par; }
 6: // call with a tainted parameter:
 7: y = func(document.cookie[0]);
 8: function count() {
 9:   return arguments.length - 1;
10: }
11: x = count(0, document.cookie[0]);
```

Figure 5. Function tainting.

```
1: document.getElementById("testtag").innerHTML =
     document.cookie;
2: var dut = document.getElementById("testtag").innerHTML;
3: // dut is tainted
```

Figure 6. DOM tree example.

to 3 due to the tainted condition on Line 1. Everything that is done within or returned by a tainted function is also tainted. When called with tainted actual parameters, the corresponding formal parameters of the function are tainted. On Line 7, the function `func` is called with a tainted actual parameter, which results in a tainted formal parameter (`par`) on Line 5. This tainted parameter is returned, and, because of this, the result of `func` on Line 7 is tainted as well. Lines 9 and 11 in Figure 5 show that `arguments.length` is tainted if one of the arguments is tainted. The second parameter on Line 11 is tainted, and therefore, the returned value on Line 9 is tainted, which results in a tainted variable `x` on Line 11.

The `eval` function is special because its argument is treated as a JavaScript program and executed. If `eval` is called in a tainted scope or if its parameter is tainted, a scope around the executed program is generated, and we conservatively taint every operation in this program.

3.2.5 Document Object Model (DOM) Tree

An attacker could attempt to remove the taint status of a data element by temporarily storing it in a node of the DOM tree and retrieving it later (see Figure 6). To prevent laundering of data through the DOM tree, taint information must not get lost when leaving the JavaScript engine. To this end, the object that implements a DOM node is tainted every time a JavaScript program stores a tainted value into this node. When the node is accessed later, the returned value is tainted.

4 Static Data Tainting

The main strength of the dynamic approach described so far is that it is capable of tracking the flow of sensitive values through data dependencies in an efficient and precise way. Unfortunately, as discussed in [7] and [27], dynamic

techniques cannot be used for the detection of *all* kinds of control dependencies. For example, consider the example attack script shown in Figure 7. This script exploits an *indirect* control dependency. On Lines 1 and 2, the variables x and y are both initialized to false. On Line 3, the attacker tests the user's cookie for a specific value. First, let us assume that the attacker was lucky, and that the user's cookie indeed holds the tested value abc. In this case, Line 4 is executed, setting x to true. At the same time, our modified JavaScript engine taints x. Variable y keeps its false value, since the assignment on Line 6 is not executed. Also, y does *not* get tainted: Remember from Section 3.2.3 that even though the generated scope covers both branches of the if construct (Lines 3 to 7), dynamic tainting occurs only along the branch that is actually executed. Since y has not been modified, this means further that the condition on y (Line 11) evaluates to true. As y is not tainted, no tainted scope is generated for this if construct, and the attacker is free to issue a request at this point in the program, carrying the information that the cookie holds the exact value abc. Analogously, in case of the more likely event that the attacker does not guess the exact cookie value, he can at least send a request indicating that the cookie does *not* hold this value, again leaking sensitive information. A more sophisticated script for cookie stealing would, for instance, employ a binary search on the cookie value instead of direct equality tests. The reason why this attack is able to circumvent dynamic protection techniques is that there exists a correlation between x and y that is encoded into the control flow. If some condition on the cookie value holds, x is set to true, while y remains false, whereas otherwise, x is false and y is true. In either case, only *one* of these variables is tainted dynamically, and hence, the other, untainted variable can be used to leak information. To cover both direct and indirect control dependencies, all possible program paths in a scope need to be examined. Unfortunately, this cannot be provided by purely dynamic methods. Therefore, to guarantee that no information can be leaked using indirect control dependencies (that is, to provide a *noninterference* [8] guarantee), static analysis is necessary. The static analysis must ensure that all variables that *could* receive a new value on any program path within the tainted scope are tainted. This is necessary because " any variable or data structure must be assumed to contain confidential information if it might have been modified within [a tainted scope]" [27]. Using both static and dynamic analysis, we can combine the strengths of both techniques to achieve full protection against XSS attacks.

4.1 Linear Static Taint Analysis

The basic idea of our static analysis is the following: For every branch in the control flow that depends on a tainted value (i.e., for every tainted scope), we have to statically analyze this scope, since dynamic analysis will only cover those parts that are executed. This static analysis must make sure that all variables that are assigned values (no matter whether these values are tainted or not) inside such a scope are also tainted. For instance, in the previous example from Figure 7, this would mean that *both* x and y are tainted, independent of the actual branch that is executed. This makes it impossible for an attacker to extract information about sensitive values without triggering an XSS alert prompt. To this end, we perform a simple, but effective *linear* static pass through the bytecode of the tainted scope. Since it is irrelevant for the analysis whether a variable is assigned a tainted or an untainted value, it is not necessary to employ a flow-sensitive analysis that understands the actual control flow. All that matters is whether a variable is modified or not. For example, one of the JavaScript instructions responsible for assigning values to variables is setname. If the static analysis encounters such an opcode during its linear pass through the tainted scope, it taints the corresponding variable (which is given as an argument to setname). If a function call or an eval statement is encountered, the JavaScript engine is switched into a special *conservative mode* where every subsequent executed instruction is considered as being part of a tainted scope. The reason is that a precise *inter-procedural* analysis would be prohibitively expensive in a real-time browser setting. By switching into a conservative mode, we prevent these additional costs, and at the same time, provide security for the user. As shown in our experiments in Section 7, this decision turned out to be feasible in practice, as only a small number of false warnings is generated.

```
 1: x = false;
 2: y = false;
 3: if (document.cookie == "abc") {
 4:   x = true;
 5: } else {
 6:   y = true;
 7: }
 8: if (x == false) {
 9:   // Line 6 was executed, and x is not tainted
10: }
11: if (y == false) {
12:   // Line 4 was executed, and y is not tainted
13: }
```

Figure 7. Attack using indirect control dependency.

One difficulty for a linear static analysis is that the instructions responsible for setting object properties (and array elements) do not specify the target object (or array) as immediate arguments because of the stack-based nature of the JavaScript interpreter. Instead, these instructions retrieve their target from the stack. As a result, in order to determine the target of an assignment to an object property,

static analysis requires information about the possible stack contents at that point in the program. To this end, static taint analysis has to be supported by an auxiliary *stack analysis*.

4.2 Stack Analysis

The purpose of stack analysis is to determine, for every program point in the analyzed scope, which elements the stack may contain. To achieve this, we employ a flow-sensitive, intra-procedural data flow analysis [2]. For each analyzed operation, we simulate the effects of this operation on the real stack by modifying an *abstract stack* accordingly. For instance, the `false` opcode is modeled by pushing an element on the abstract stack that represents a boolean value. Note that it is not necessary to track the exact boolean value, it is sufficient to know that there is *some* boolean value on the stack. For objects and arrays, however, the stack content is modeled in more detail, so that it is possible to determine the target objects (and arrays) of assignment instructions. The data flow analysis is greatly simplified by the fact that the JavaScript engine demands the size of the stack to be the same at control flow merge points, regardless of the actual program path taken. This way, the fixpoint iteration algorithm of data flow analysis terminates quickly, and since stacks cannot grow infinitely during loop constructs, there is no need for a widening operator to enforce termination [24].

Currently, not all bytecode instructions are modeled in our implementation. For instance, more complex operations such as `throw` or `try` have been omitted. To achieve safe results in spite of this limitation, the stack analysis informs the static taint analysis whenever such an instruction occurs in the analyzed scope. Subsequently, the static taint analysis safely assumes that *all* variables (and objects) that are loaded onto the stack in this scope will be the target of an assignment, and taints them as a result. This ensures that the attacker is not able to leak information due to unmodeled instructions, keeping the user secure.

4.3 Justifying Hybrid Analysis

As discussed in the previous sections, our approach to XSS prevention is to apply dynamic analysis techniques in general, and static analysis techniques only when it is necessary. An apparent alternative to this technique would be to perform only static analysis. However, the reasons for using a hybrid analysis are precision and efficiency. It is a well-known fact that dynamic analysis generates more precise results than static analysis, which suffers from the conceptual limitation of undecidability. Besides, precise static analysis techniques are computationally expensive. This might be irrelevant for static security analyses performed by application developers before the deployment of the application. However, in real-time settings, dynamic analysis techniques are more suitable. By switching to a relatively fast type of static analysis only at those points where it is necessary, we combine the best of both approaches.

5 Data Transmission

The tainting mechanisms described so far only track the status of data elements while they are processed by the JavaScript engine. No steps are taken to prevent the leakage of sensitive information. For example, the execution of JavaScript statements is not prevented in case of tainted variables, nor is any data or part of it removed during the processing. For a cross-site scripting attack to be successful, the gathered data needs to be transferred to a site that is under the attacker's control. That is, the tainted data has to be transferred to a third party. This transfer can be achieved using a variety of methods. Some examples include:

- Changing the location of the current web page by setting `document.location`.
- Changing the source of an image in the web page.
- Automatically submitting a form in the web page.
- Using special objects, such as the `XMLHttpRequest` object.

To successfully foil a cross-site scripting attack, we prevent the transfer of tainted data to third-party domains with any of these methods. More precisely, we ask the user whether this transfer should be allowed.

6 Implementation

Our prototype implementation extends the Mozilla Firefox 1.0pre [20] web browser. There are two different parts in the web browser that can contain tainted data objects. One part is the JavaScript engine, which is called SpiderMonkey [18]. Here, variables, functions, scopes, and objects can be tainted as a result of sensitive data that is processed by JavaScript programs. The other part is the implementation of the DOM tree (e.g., `location.href`). To store the additional tainting information, we modified data structures in both parts of the browser. Even though we were careful not to introduce deep changes to the program logic and tried to reuse existing facilities, it turned out that a considerable engineering effort was required to implement the modifications.

Every time a JavaScript program attempts to transfer sensitive data, a check is performed to determine whether the host from which the document is loaded and the host to which sensitive data is sent are from different domains. If

this is the case, an alert is raised, and the user can decide if the transfer should be allowed or denied. Alternatively, the user can choose to permanently allow or deny all transfers between the two domains, or to permanently allow or deny all transfers to the offending destination domain, regardless of the current source domain.

7 Evaluation

To evaluate our system, we took several complementary approaches. The most immediate step, which was conducted during the development phase of our prototype, was to perform a wide range of functionality tests by exploiting a variety of small XSS vulnerabilities. These tests were based on the experiences with cross-site scripting that our group's members have collected in the past. As expected, these tests confirmed that our concepts were indeed capable of protecting the user against XSS attacks.

In addition to tests with XSS attacks that were designed internally, we verified that our system was also suitable for defending against real-world exploits. To this end, we installed vulnerable versions of the following popular open-source web applications: phpBB 2.0.18 [25], myBB 1.0.2 [9], and WebCal 3.04 [17]. Then, we launched reported XSS attacks [1, 4, 5] against each of these applications. Again, our mechanism was successful in detecting these attacks, and in each test the user was reliably prompted before any sensitive information could be leaked.

Apart from defending against XSS attacks, in order to be useful in practice a protection scheme must be efficient and not bother its users with countless false warnings. To evaluate these aspects, we conducted both manual and automatic tests. For the manual tests, the modified browser was used by the authors for web surfing on a daily basis. Compared to the amount of processing necessary to fetch and render web pages, the overhead of our extended JavaScript engine was negligible. Also, the amount of false positives was low, although we were regularly prompted with warnings of sensitive data transfers.

Interestingly, although these alerts were not the result of XSS attacks, they correctly warned about attempts to transfer sensitive information across domain borders. These information transfers were caused by scripts from companies that provide web site statistics or that perform user tracking. These scripts gather information (URL, referrer, title, cookie, etc.) about the currently-visited page and transfer it to a web application hosted on a different domain. Of course, such information flows are not caused by cross-site scripting attacks, as the scripts are inserted into the web page with the consent of the web site owner. However, we believe that the warnings are actually useful because they provide the user with additional control over the transmission of sensitive data. This way, the user is given the chance to decide whether she regards the collection of this information as a violation of her privacy.

Our prototype permits to conveniently define persistent policies so that the user has to decide for a particular destination domain only once. For example, if an alert is generated for an information transfer from `www.slashdot.org` to `www.google-analytics.com`, the user can instruct our prototype to allow or deny transfers between these two domains forever. Alternatively, she can also decide to permanently allow or deny transfers to `www.google-analytics.com`. This reduces the number of warning prompts considerably. A higher level of convenience for the user can be achieved by pre-configuring the browser with a number of typical, harmless destination domains. For novice users, who might have problems with taking the right decisions, the decision procedure can be made completely automatic by disallowing all suspicious connection attempts. Even though this might have an impact on legitimate functionality in rare cases, we are confident that this is a reasonable and safe alternative for technically-unsophisticated users.

Even though the manual testing showed that our approach is effective and efficient, we wanted to test our solution on a more extensive set of data. Therefore, to obtain a comprehensive amount of test data, we enhanced the Firefox browser with a web crawling engine. Using this crawler, we were able to automatically visit a large number of pages and determine a more representative estimate of incorrect warning prompts that a user can expect when browsing. Note that a traditional crawler would not be suited for our needs, since it is not sufficient to simply fetch and store HTML pages. Instead, it is necessary to take embedded JavaScript code into account, and to simulate user behavior in a realistic way. Because the crawler is directly using the Firefox web browser, it is capable of interpreting JavaScript code so that our protection mechanisms are activated automatically. Moreover, to simulate user behavior and trigger JavaScript that is only activated when input is typed into form elements, the crawler fills out all encountered web forms and submits them. Finally, another important aspect is the triggering of JavaScript events. Many web pages contain JavaScript code that is executed in association with specific user actions (such as `onclick` or `onmouseover`). To achieve exhaustive coverage of code embedded in web pages, our crawler deliberately triggers these events and also continues its analysis on pages that are requested as a consequence of triggering these events.

By using our crawler, we were able to perform a large-scale empirical evaluation of our XSS protection mechanisms, visiting a total of 1,033,000 unique web pages. To achieve a broad coverage of visited domains, we limited the maximum number of pages to be visited per domain to 100.

Destination Domain	Number of Flows	Type of Domain
.google-analytics.com	35,238	tracking, web statistics
.2o7.net	11,404	tracking, web statistics
.hitbox.com	6,458	tracking, web statistics
.webtrendslive.com	3,196	tracking, web statistics
.statcounter.com	2,518	tracking, web statistics
.sitemeter.com	2,099	web statistics
.revsci.net	1,866	tracking, advertisement
.blogger.com	1,221	blogging service (tracking)
.statistik-gallup.net	1,119	web statistics, tracking
.sitestat.com	899	tracking, web statistics
.gemius.pl	835	web statistics
.webtrends.com	690	tracking, web statistics
.urchin.com	662	web statistics, tracking
.liveperson.net	533	web statistics
.intellitxt.com	502	advertisement
.atdmt.com	470	tracking, advertisement
.tribalfusion.com	466	advertisement
.espotting.com	438	advertisement
.monster.com	430	career network (tracking)
.coremetrics.com	382	web statistics, tracking
.realmedia.com	363	tracking, web statistics
.hitslink.com	360	web statistics
.kontera.com	354	advertisement
.adbrite.com	339	advertisement
.akamai.net	330	web statistics, tracking
.247realmedia.com	316	advertisement
.estat.com	296	tracking, web statistics
.seeq.com	296	advertisement
.questionmarket.com	278	advertisement
.netflame.cc	267	tracking, web statistics

Table 2. Top-30 destination domains that caused the majority of the alert prompts.

From all visited pages, 88,589 (8.58%) triggered an XSS alert prompt. After a closer inspection of these warnings, it turned out that a majority of them were caused by attempted connections to only a few destination domains. Just as we expected from our manual experiments, these domains belong to companies that collect statistics about traffic on the web sites of their customers. Table 2 lists the top 30 domains that were the target of most information flows, the number of flows to these domains, and the types of companies that own them. When providing rules (deny or accept) for only these top 30 domains, it is possible to reduce the number of alert prompts to 13,964 (1.35%). For instance, this could be achieved by shipping the enhanced browser with a built-in list of these domains, and by denying the transfer of sensitive information to these domains by default. If the user has fewer concerns about privacy, she can still change some or all of these rules into accept rules.

A further reduction of the number of alert prompts can be achieved by being less restrictive about what kind of data is considered to be sensitive. Our current implementation is rather restrictive in this respect, and even protects less critical pieces of data such as `document.lastModified`. Usually, the sole information that has to be protected in order to foil XSS attacks is information stored in cookies. By analyzing those remaining alerts that were not caused by the Top-30 domains mentioned above, it turned out that only 5,289 of these alerts were due to attempts to transfer cookie data. This means that by focusing on the protection of cookies, the number of alert prompts can be further reduced from 13,964 to 5,289 (this value amounts to one prompt for every two hundred random pages that are visited). A more detailed breakdown of the different causes for alert prompts can be found in Table 3. Note that some prompts are the result of more than one sensitive source.

Sensitive Source(s)	Information Flows
Cookie	5,289
Form Data	735
Location	8,187
Referrer	8,696
Title	4,246
Links and Anchor	171
Status	726

Table 3. Sensitive information transferred to the remaining domains (not Top-30).

After inspecting a small sample of the 5,289 cases responsible for cookie-related alert prompts, it turned out that in many of these cases, user information was sent to less-known tracking sites that happened to be not in the Top-30 list. Another group of warnings were "semantic" false positives, in the sense that even though cookie information was transferred to a different domain, it was not

transferred across company borders. For instance, we observed an exchange of sensitive data between cnn.net and cnn.com. In a less obvious case, data transfer took place between the domains discover.com and unitedstreaming.com, which turned out to belong to the same company. Finally, we also observed some false positives that were due to our conservative tainting approach. For example, some pages use JavaScript to check whether the browser allows cookies to be set. To this end, the script first stores some string into a cookie and immediately reads it back. Then, a check is made to determine whether the value was successfully stored. Because the cookie is considered sensitive, this check opens a tainted scope. As we want to prevent information leaks that exploit indirect control flows, all values written in this scope have to be tainted. When one of these values is later used in a cross-domain connection, a warning is raised.

In summary, the results of our empirical evaluation demonstrate that only a small number of false warnings is generated. Besides, even though these warnings do not correspond to real XSS attacks, they still provide the user with additional control in terms of web privacy. Given that our protection approach provides strong security against cross-site scripting, we believe that our system is a practical and viable solution against XSS attacks.

8 Conclusions

Cross-site scripting (XSS) is one of the most frequent vulnerabilities found in modern web applications. Nevertheless, many service providers are either not willing or not able to provide sufficient protection to their users. This paper proposes a novel, client-side solution to this problem. By modifying the popular Firefox web browser, we are able to dynamically track the flow of sensitive values (e.g., user cookies) on the client side. Whenever such a sensitive value is about to be transferred to a third party (i.e., the adversary), the user is given the possibility to stop the connection. To ensure protection against more subtle types of XSS attacks that try to leak information through non-dynamic control dependencies, we additionally employ an auxiliary, efficient static analysis, where necessary. With this combination of dynamic and static techniques, we are able to protect the user against XSS attacks in a reliable and efficient way. To validate our concepts, we automatically tested the enhanced browser on more than one million web pages by means of a crawler that is capable of interpreting JavaScript code. The results of this large-scale evaluation demonstrate that only a small number of false positives is generated, and that our underlying concepts are feasible in practice.

References

[1] addmimistrator@gmail.com. MyBB 1.0.2 XSS Attack in search.php Redirection. http://www.securityfocus.com/archive/1/423135, January 2006.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] J. Allen. Perl Version 5.8.8 Documentation - Perlsec. http://perldoc.perl.org/perlsec.pdf, 2006.

[4] M. Arciemowicz. phpBB 2.0.18 XSS and Full Path Disclosure. http://archives.neohapsis.com/archives/fulldisclosure/2005-12/0829.html, December 2005.

[5] S. Bubrouski. Advisory: XSS in WebCal (v1.11-v3.04). http://archives.neohapsis.com/archives/fulldisclosure/2005-12/0810.html, December 2005.

[6] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2004.

[7] D. E. Denning. A Lattice Model of Secure Information Flow. In *Communications of the ACM*, 1976.

[8] J. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, 1982.

[9] M. Group. MyBB - Home. http://www.mybboard.com/, 2006.

[10] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Twenty-First Annual Computer Security Applications Conference (ACSAC)*, 2005.

[11] O. Hallaraker and G. Vigna. Detecting Malicious JavaScript Code in Mozilla. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS05)*, 2005.

[12] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. In *Proceedings of the 18th International Conference on Advanced Information Networking and Application (AINA04)*, March 2004.

[13] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[14] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *The 21st ACM Symposium on Applied Computing (SAC 2006)*, 2006.

[15] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *10th ACM Conference on Computer and Communication Security (CCS-03) Washington, DC, USA, October 27-31*, pages 251 – 261, October 2003.

[16] G. D. Lucca, A. Fasolino, M. Mastroianni, and P. Tramontana. Identifying Cross Site Scripting Vulnerabilities in Web Applications. In *Sixth IEEE International Workshop on Web Site Evolution (WSE'04)*, pages 71 – 80, September 2004.

[17] marndt@bulldog.tzo.org. WebCal - A Web Based Calendar Program. http://bulldog.tzo.org/webcal/webcal.html, May 2003.

[18] Mozilla Foundation. SpiderMonkey - MDC. http://developer.mozilla.org/en/docs/SpiderMonkey, December 2005.

[19] Mozilla Foundation. JavaScript Security: Same Origin. http://www.mozilla.org/projects/security/components/same-origin.html, February 2006.

[20] Mozilla Foundation. Mozilla.org - Home of the Mozilla Project. http://www.mozilla.org, 2006.

[21] Netscape. Using data tainting for security. http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/advtopic.htm, 2006.

[22] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *20th IFIP International Information Security Conference*, Makuhari-Messe, Chiba, Japan, 05 06 2005.

[24] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[25] phpBB Group. phpBB.com :: Creating Communities. http://www.phpbb.com, 2006.

[26] T. Pietraszek and C. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[27] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. In *IEEE Journal on Selected Areas in Communications*, pages 5 – 19, January 2003.

[28] G. Suh, J. Lee, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[29] W3C - World Wide Web Consortium. Document Object Model (DOM) Level 3 Core Specification. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/DOM3-Core.pdf, April 2004.

[30] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *15th Usenix Security Symposium*, 2006.