Bypassing XSS Auditor: Taking Advantage of Badly Written PHP Code

Anastasios Stasinopoulos, Christoforos Ntantogian, Christos Xenakis
Department of Digital Systems, University of Piraeus
{stasinopoulos, dadoyan, xenakis}@unipi.gr

Abstract*—XSS attacks have become very common nowadays, due to bad-written PHP web applications. In order to provide users with rudimentary protection against XSS attacks most web browser vendors have developed built-in protection mechanisms, called XSS filters. In this paper, we analyze two attacks that take advantage of poorly written PHP code to bypass the XSS filter of WebKit engine named XSS Auditor and perform XSS attacks. In particular, the first attack is called PHP Array Injection, while the second attack is a variant of the first one and it is named as PHP Array-like Injection. Both attacks take advantage of improper management of variables and arrays in PHP code to bypass the XSS Auditor. We elaborate on these attacks by presenting concrete examples of poorly written PHP code and constructing attack vectors to bypass the XSS Auditor. To defend against the identified attacks, we provide proper code writing rules for developers, in order to build secure web applications. Additionally, we have managed to patch the XSS Auditor, so that it can detect our identified XSS attacks.*

***Keywords*—** *Cross Site Scripting, XSS, WebKit, XSS Auditor, PHP Array Injections, Quote-Jacking*

I. INTRODUCTION

PHP (a recursive acronym for "PHP Hypertext Preprocessor") is an open source server-side scripting language designed for web development, but it is also used as a general-purpose programming language. According to Netcraft's Web Server Survey [1], by January 2013, PHP was installed on more than 240 million websites. The most prominent websites that use PHP include Google, Facebook, Yahoo!, Wikipedia, Amazon, Ebay, YouTube, Flickr, and many more. PHP code can be mixed with HTML code or it can be used in combination with various template engines and web frameworks. PHP code is usually processed by a PHP interpreter, which is implemented as a web server's native module or a Common Gateway Interface (CGI) executable. After the PHP code is interpreted and executed, the web server sends the resulting output to its client, usually as a part of the served web page.

Although PHP is a powerful, free, and easy to learn and use programming language, it comes with certain features that makes easy to write insecure code. According to the National Vulnerability Database [2], in 2013, 9% of all vulnerabilities reported were related to PHP [3]. Furthermore, it is worth noting that since 1996 about 30% of all vulnerabilities, which are reported to the same database are related to PHP. Web applications that are implemented in PHP can be vulnerable to various exploit vectors, such as XSS (Cross-Site Scripting), SQL Injections, CSRF (Cross-Site Request Forgery) injections etc. The OWASP Top Ten for 2013 [4] lists XSS as the most common security risks to web applications. More specifically, XSS [5] is an application-layer threat that emanates from the security weaknesses of client-side scripting languages, HTML and JavaScript (or more rarely VBScript, ActiveX or Flash). The purpose of an XSS attack is to inject malicious code in a website, in order to bypass security access controls and compromise data or perform session hijacking. An XSS attack occurs when an adversary manipulates and injects a malicious code in a web application (usually JavaScript), in order to alter data contexts in HTML code into a scripting context -usually by injecting new HTML, JavaScript strings or CSS markup. This injection code is sent to the web application via HTTP parameters and it is executed by the client browser and eventually, inserted into the output of the web application. There are three categories of XSS vulnerabilities:

a) Reflected XSS: The concept of this kind of XSS attack is that the victim clicks on a crafted link and the attack is initiated. More specifically, an XSS vulnerability is reflected in the application's output, if the injection is echoed by the server in the response of an HTTP request. Reflection can occur with error messages, search engine submissions, comment previews, etc. This form of attack can be mounted by persuading a user to click a link or submit a form of the attacker's choice, issues that may involve emailing the target, mounting a UI Redress attack, or using a URL shortener service to disguise the URL.

b) Stored XSS: The injection is resilient throughout sessions by being permanently stored in a data storage and it is echoed every time a user visits the unsafe web site or views the targeted data. Obviously, the range of potential victims is greater than in the reflected XSS, since the payload is displayed to any visitor.

c) DOM-Based XSS: DOM-based XSS attacks control the web page's Document Object Model (DOM), which serves as a cross-platform and a language-independent model that interacts with objects in HTML. DOM-based XSS can be either reflected or stored. The attacker is allowed to run JavaScript scripts in a web browser through targeting vulnerabilities in the HTML code and interacting with the DOM of the web page.

The current countermeasures to detect XSS attacks are: i) server-side, ii) network-based using Web Application Firewalls (WAF), and, iii) client-based using XXS filters at the level of browsers. In this work, we will specifically focus on XSS filters and how can be trivially bypassed using simple yet effective techniques. By design, web browsers must execute the HTML and JavaScript code which is obtained from a web application through the HTTP protocol. The objective of XSS filters is to detect "dangerous" attribute tags (i.e., <img>, <script>, etc.) inside HTTP parameters of the GET and POST HTTP methods and prevent the execution of the injected JavaScript code. An XSS filter, named XSS

978-1-4799-1812-6/14/$31.00 ©2014 IEEE

auditor [6] has been implemented in the WebKit browser engine [7] and it is used by the most widely used browser, that is Google Chrome / Chromium as well as by the Apple Safari browser. Microsoft's Internet Explorer browser, from version 8 to the latest, provides an XSS filter [8]. On the other hand, Mozilla Firefox does not include a pre-installed XSS filter, but there is a free plugin named NoScript [9], which can be installed by end-users and it is considered to be an efficient XSS filter for Firefox browser. Each of the above XSS filters detects XSS attack vectors with different techniques. More specifically:

a) The XSS filter of Internet Explorer handles regular expressions to identify malicious attack vectors in outgoing HTTP requests. The filter creates a unique signature of the potentially malicious part, instead of removing it, and waits for the HTTP response to arrive at the web browser. If the signature matches anything that is contained inside the response, the filter blocks and eliminates the suspicious parts.

b) NoScript Firefox Plugin handles regular expressions to identify outgoing HTTP requests for malicious attack vectors. If there is a match discovered between the regular expressions and the parts of the URL concerning the attack vector, then these URL parts are removed from the HTTP request.

c) Unlike the previous two filters, XSS Auditor does not use regular expressions to filter outgoing HTTP requests [10]. In particular, the XSS Auditor examines the DOM tree created by the HTML parser, making the semantics of those bytes clear. In this way, the XSS Auditor can easily identify which parts of the response are being treated as script.

We have pinpointed that badly written web application has the unfortunate ability of disabling XSS filters. Thus, developers of web applications may unintentionally help malicious actors to perform XSS attacks. In this paper, we analyze two attacks that take advantage of poorly written PHP code to bypass the XSS filter of WebKit (i.e., XSS Auditor) and perform XSS attacks. In particular, the first attack is called PHP Array Injection, while the second attack is a variant of the first one and it is named as PHP Array-like Injection. Both attacks exploit improper use or management of variables and arrays in PHP code to bypass the XSS Auditor. We elaborate on these attacks by presenting concrete examples of poorly written PHP code and constructing attack vectors to bypass the XSS Auditor. We have also audited the source code of PHP applications to examine the prevalence of the identified attacks. We have discovered that many open-source Content Management Systems (CMS) are vulnerable to PHP Array-like Injection attacks. Finally, to defend against the identified attacks, we provide proper code writing rules for developers, in order to build secure web applications. Additionally, we have managed to patch the XSS Auditor, so that it can detect our identified XSS attacks.

The rest of the paper is organized as follows. Section 2 provides the background analyzing the related work and the architecture of the XSS Auditor. Section 3 present examples of badly written PHP code that may result in bypassing XSS Auditor and elaborates on the identified XSS attacks. Section 4 provides secure code writing guidelines and analyzes the patches that we have committed to XSS Auditor. Finally, Section 5 includes the conclusions.

II. BACKGROUND

A. Related Work

The literature includes many works that analyze successful attempts of bypassing the XSS Auditor and XSS filters in general. In this section, we mention the most prominent works, since a comprehensive analysis of all the related literature requires an extensive review, which is outside the scope of this paper. When the XSS Auditor was applied for the first time in Google Chrome, a series of bypasses took place by the sla.ckers.org forum [11]. Moreover, in [12], it was proved that XSS Auditor can be bypassed using two or more parameters [13]. In [14], the XSS Auditor was bypassed using <svg> tags and html-entities, while in [15] the authors used the "U+2028" and "U+2029" Unicode characters to bypass XSS auditor. Note the attacks in [14] and [15] have been fixed in the new versions of XSS Auditor. In the most recent work [16], the authors discovered a whopping seventeen security flaws that allowed them to bypass the XSS Auditor's filtering capabilities.

Apart from XSS auditor, researchers have successfully bypassed other XSS filters. In [17], the NoScript plugin for Firefox browser was bypassed through an Error Based SQL injection flaw, while in [18] it was proved that the XSS filter of Internet Explorer can be easily bypassed by taking advantage of techniques that turn injected untrusted data into trusted data, which is not subject to validation by Internet Explorer's XSS filter. Recently, a new class of XSS attacks was discovered named as mutation-based XSS (mXSS) [19] that may occur in innerHTML and related DOM Javascript properties. The mutation-based XSS (mXSS) attack vectors affect all three major browsers (i.e., Chrome, Firefox and MS Inter Explorer). As a matter of fact, mutation-based XSS (mXSS) vectors are not limited only to client-side XSS filters, but can be used to successfully bypass widely deployed state-of-the-art server-side XSS protections mechanisms, including Web Application Firewall (WAF) systems and Intrusion Detection and Intrusion Prevention Systems (IDS/IPS).

Unlike the previously mentioned works that elaborate on techniques and methods to bypass XSS protection mechanisms, in this paper we focus on various mistakes that PHP web application developers make and unwittingly help the attackers to bypass the XSS Auditor and cause harm the end-users.

B. XSS Auditor

XSS Auditor is placed between the HTML parser, (a component of Rendering Engine which is responsible for parsing the HTML into a tree (parse tree) of DOM element and attribute nodes) and the JavaScript Interpreter (a virtual machine which interprets and executes JavaScript code) as shown in Figure 1. It is worth noting, that different web browsers use different rendering engines. For example, Internet Explorer uses Trident, Firefox uses Gecko, Safari

uses WebKit, while Chrome and Opera (since version 15) use a fork of WebKit named Blink [20].

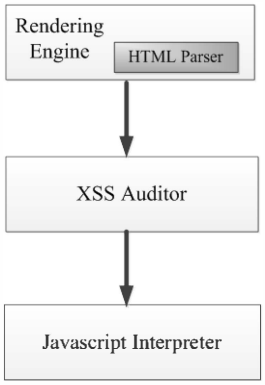

Fig. 1. XSS Auditor is between the HTML parser and the JavaScript Interpreter.

An important characteristic of XSS Auditor is that it inspects only GET / POST HTTP responses [10]. If the same executable JavaScript is detected in both HTTP request and response, the XSS Auditor raises an alert and prevents the injected script from being executed. More specifically, the auditing process consists of three parts:

a) Firstly, the XSS Auditor checks for "dangerous" event attributes, that either contain a JavaScript URL or have the name of an HTML event handler (i.e onerror, onclick, onload, etc.). A JavaScript can be executed when an event occurs. Thus, if such an HTML event attribute is found, the XSS Auditor checks the corresponding HTTP request and in case a match is found, the filter assumes that the event attribute is malicious and deletes the attribute value.

b) Secondly, the XSS Auditor performs tag-specific checks for "dangerous" event attributes. Note that except for attributes containing JavaScript URLs or attributes that have the name of an HTML event handler, there are also other attributes HTML tags, such as <script>, <object>, <param>, <embed>, <applet>, <iframe>, <meta>, <base>, <form>, <input> and <button> that need to be filtered.

c) Thirdly, the XSS Auditor, filters injected inline scripts. Whenever the XSS Auditor identifies a script tag, validates the content that is enclosed between the opening and the closing script tag. Assuming that an injection has occurred and the content has been found in the request, it will result in replacing the content with an empty string.

III. Bypassing XSS Auditor

In this section, we elaborate on the discovered XSS attacks that bypass the XSS Auditor. In all experiments, the aim is to execute the Javascript code <script>alert(1)</script>. In other words, the attack vector (i.e., the inserted code <script>alert(1)</script>) should be parsed by the HTML parser and then to be transferred to the JavaScript Interpreter for execution. To perform the attacks, we have developed in PHP web applications with subtle mistakes that result in bypassing XSS Auditor. The injection point of the Javascript code is the URL of the vulnerable PHP web applications. If the attack is successful, an alert box is prompted that simply includes the text "1". We also mention that the experiments performed in the most recent versions of the browsers that use XSS Auditor. That is, Google Chrome version "36.0.1985.143 m" for Windows OS and version "36.0.1985.143" for Linux OS, Google Chromium version "35.0.1916.153 Built" for Linux, Apple Safari version "6.1-7537.71" for Mac OS X 10.7.5.

A. The PHP Arracy Injection Attack

In this section we describe our identified attack named as PHP Array Injection. This attack can be performed in poorly written PHP code and specifically when web applications use the print_r() or var_export() (see figure 2) PHP functions to print back the name of the super global array $_GET and the value of it. In particular, the PHP Array Injection attack can bypass the XSS Auditor when an adversary has under his/her control two URL variables of a web application (that uses the above PHP functions), due to the use of arrays which can hold more than one value at a time. Note that the use of the print_r() and var_export() functions is common in web applications, because they offer a simple way to process (e.g., print, debug, etc.) a super global variable (e.g., $_GET) using arrays.

Moreover, to demonstrate this attack, we have developed for testing purposes a simple web application that implement the code snippet shown in figure 2 (i.e., snippet 1). Note that the URL of the testing web application, which we performed the experiments is: "*http://localhost/xssme/index.php?x[Key]=value*". The testing web application allows any variable that is an input as GET data to be assigned to any variable name that a user defines. Thus, an adversary can inject his/her own Javascript code. Moreover, in this application, the PHP Array Injection attack exploits the fact that the URL allows a user to have under his/her control both the *Key* and the *Value* variables. Additionally, using the print_r() and var_export() functions, an attacker can print back a chosen key of the "*x[ ]*" array and the value of it. Finally, it is important to notice that the snippet 1 uses the htmlspecialchars() function to escape output data (see figure 2). However, as we analyze in section 4, the use of this function alone cannot prevent XSS attacks.

In our first attempt of the attack, we tried to replace in the above URL, either the "Key" or "Value" with the XSS attack vector "*<script>alert(1);</script>*", as shown in the following two examples:

x[<script>alert(1)</script>]=Value

and,

x[Key]=<script>alert(1)</script>

However, the previous two attempts were unsuccessful, because the XSS Auditor can trivially identify the inserted Javascript code. Based on [12], we realized that XSS Auditor is not designed to detect injections, which are split across multiple parameters. Thus, we perceived that we need to split the attack vector: *"<script>alert(1);</script>"* into two or more parts and then make the JavaScript engine ignore the text "]=" between the controlled parts. Initially, we thought

that this can be achieved using the JavaScript multi-line comment delimiters '/* */' because any text between '/*' and '*/' will be ignored by JavaScript. For example, the first attack vector part can be defined as *x[<script>alert(1);* and the second part as */*]=*/</script>*. Hence, we tried to execute the attack vector:

x[<script>alert(1);/]=*/</script>*

However, this attempt was again unsuccessful. After an extensive search in the source code of XSS Auditor, we discovered that the latter prevents every attempt that involves HTML and JavaScript comments, such as '<!-- -->' , '//' and '/* */'. If either the URL or the request body contains comment characters, the filter is activated and the attempt was blocked.

```
<?php

  //---------------------------------------------------
  // Case 01 - The "print_r()" function.
  //---------------------------------------------------
  echo '<li><b>The "print_r()" function response: </b><br></li>';
  $get = print_r($_GET['x']);
  echo htmlspecialchars($get, ENT_QUOTES, 'UTF-8');

  //---------------------------------------------------
  // Case 02 - The "var_export()" function.
  //---------------------------------------------------
  echo '<br><br><b> <li>The "var_export()" function response:
</b><br></li>';
  $get = var_export($_GET['x']);
  echo htmlspecialchars($get, ENT_QUOTES, 'UTF-8');

?>
```

Fig.2. Snippet 1 which is vulnerable to PHP Array Injection attack.

1) The Quote-Jacking Technique

After our initial failed attempts, we used a technique that we named it Quote-Jacking to perform the PHP array injection attack and successfully bypass the XSS Auditor. In particular, we tried to repeat the previous attempts but this time we replaced the aforementioned comment characters with double-quotes (**""**) or single-quotes (**''**), in order to comment out the second parameter. The rationale behind the Quote-Jacking technique is that any string between <script> and </script> tags, which is enclosed by single-quotes or double quotes, is treated as a comment and should be ignored when a JavaScript function (e.g., alert()) is executed. The attack vector based on the Quote-Jacking technique is:

x[<script>alert(1);"]="</script>

Using the PHP Array Injection combined with the Quote-Jacking technique we were able to successfully bypass the XSS Auditor.

It is worth noting that the PHP array injection combined with the Quote Jacking technique can be performed by replacing in the above attack vector the semicolon character ';' with any element from Table 1. For example, the following attack vector can also bypass the XSS filter:

x[<script>alert(1)"]="</script>*

This happens because during our experiments we observed that any NaN (Not-a-Number) [21] result from a Javascript code execution, leads to successful XSS Auditor evasion. By introducing any operator or element from the table we achieve to perform an operation with NaN result and bypass the XSS Auditor. It is important to mention that this finding is not limited only to Javascript code that includes the alert() function, but on the contrary it can be generalized for any Javascript function without a return statement, or a function with an empty return statement which gives as a result "undefined".

Table 1. Operators and elements for the Quote-Jacking technique.

JavaScript Operators	
1. Assignment operators	**+=, -=**, *=, /=, %=, |=, ^=, >>=, <<=, >>>=
2. Comparison operators	==, ===, !=, !==, >, >-, <, <=, >=
3. Arithmetic operators	***, %, -**, /
4. Bitwise operators	|, ^, ~, >>, <<, >>>
5. Logical operators	||
JavaScript Functions	
1. Functions	void(), new(), typeof(), this(), delete(), in(), instanceof()
Other accepted characters	
1. Parenthesis / Brackets / Braces	(), [], {}
2. Semicolon / linefeed Character / carriage return Character	;, %0a, %0d

B. The PHP Array-like Injection Attack

Except for the vulnerable PHP code that we mentioned in the previous section, we have pinpointed that the XSS Auditor can be bypassed with other poorly-written PHP code. For instance, consider the PHP code snippet shown in figure 3 (i.e., snippet 2). This code uses a "foreach" loop to print keys and values of a $_GET super global array. This code is an alternative way to produce the same results as snippet 1, but without making a use of **print_r()** or **var_export()** functions. Similarly, figure 4 shows another example of badly written code snippet (i.e., snippet 3) that has exactly the same functionality as snippet 1, but does not use arrays.

```
<?php
foreach($_GET as $key => $value){
  echo "The key ".$key." has the value ".$value." <br>";
}
?>
```

Fig. 3. Snippet 2 which is vulnerable to PHP Array-like Injection attacks.

```
<?php
  $key = key($_GET);
  $value = $_GET[$key];
  echo "The key ".$key." has the value ".$value." <br>";
?>
```

Fig. 4. Snippet 3 which is vulnerable to PHP Array-like Injection attacks.

The above two vulnerable PHP code snippets can help an attacker to bypass the XSS Auditor by performing a variation

of the PHP array injection attack, named PHP Array-like Injection. This attack targets against PHP applications which behave like making use of arrays but they don't make actual use of them. For example, in figure 3 snippet 2 does not explicitly use arrays but instead uses the "foreach" loop to print keys and values of a $_GET array. Again, to demonstrate this attack we have developed in PHP testing applications that implement the above snippets. The developed applications use the URL: "http://localhost/xssme/index.php?a=b". An attacker via the $_GET variable can insert a custom name for a key (i.e,. *'a'* in the URL) and a corresponding value (i.e., *'b'* in the URL). If we use in the above URL as key the string "*<script>alert(1);*" and as value the string "*</script>*" then we achieve a NaN result and bypass the XSS Auditor as analyzed previously (see section 3.1.1). The URL with the final attack vector is

<script>alert(1);"="</script>

C. Impact

To examine the prevalence of the identified attacks, we have audited the source code of various PHP applications. We have discovered that many open-source CMS are vulnerable to PHP Array-like Injection attacks. In particular, we have pinpointed that the file and image manager plugin "Ajax File Manager v1.0" used by Tinymce and FCKeditor editors in the file named ajax_create_folder.php is vulnerable to the PHP Array-like Injection attacks. This plugin is used in various applications such as "Ajax File Manager", XOOPS 2.5.0-2.5.4, OSClass 3.4.3, Zenphoto 1.4.1.4, phpMyFAQ <= 2.7.0, PrestaShop 1.5, A6-CMS (ACMS) 5.30, Log1 CMS 2.0 and many more. In the new versions of these applications the vulnerable plugin has been corrected.

It is evident that a successful exploitation and bypass of a browser-based XSS filter (including XSS Auditor) could lead to an XSS attack to every user that visits the vulnerable web application. The impact of XSS attacks if often misconceived by developers, because they consider that XSS attacks cannot be exploited to steal personal data of end-users. In our point of view, the consequences of XSS attacks can be devastating. In particular, an attacker exploiting XSS vulnerabilities can perform several malicious actions including:

a) Steal or take over a user's session (i.e., session hijacking).

b) Monitor a user's activities.

c) Steal sensitive data from the user's browser on a personal computer, smart-phone or tablet.

d) Execute arbitrary code on the user's personal computer, smart-phone or tablet.

e) Take full control of the user's personal computer personal computer, smart-phone or tablet

f) Pivot and attack the network(s) connected to the user's personal computer, smart-phone or tablet.

IV. COUNTERMEASURES

The best strategy to prevent XSS attacks is the adherence to secure coding guidelines and practices, in order to build secure applications without vulnerabilities. In this section we provide some indicative secure coding practices against our identified attacks. A full detailed guide to prevent XSS attacks is also available on OWASP [22]. Additionally, we present the procedure that we followed to patch and enhance the security of the XSS Auditor.

A. HTML Escaping

The most important rule against XSS attacks (and in general injection attacks) is "Filter Input - Escape Output". More specifically, by escaping data on output we ensure that data cannot be misinterpreted by the parser or interpreter. The obvious examples are the "<" and ">" characters that denote element tags in HTML. If these characters were allowed to be inserted in a user-supplied input, it would allow an attacker to introduce new tags (i.e. <img>, <script> etc.) that the browser would render. For this reason, developers should escape these special characters by using htmlspecialchars() PHP function [23]. This function, apart from "<" and ">" that have already been mentioned, also converts other special characters, such as "&", "'" and "" to HTML entities. Escaping can be performed using the PHP funcation htmlspecialchars() which should be called with the ENT_QUOTES flag and a charset parameter. The ENT_QUOTES flag specifies how double and single quotes should be handled. Without passing ENT_QUOTES as the second parameter, single-quote chars are not encoded. In summary, the presented snippets should properly use the htmlspecialchars() function to avoid PHP Array-like Injection attacks as shown in figure 5.

Moreover, it is important to ensure that the web application specifies the character encoding for the HTML document as UTF-8 character-set in a header() function, or in a <meta> tag at the beginning of the <head> element. The <meta> tag provides meta-data about the HTML document. With this specification, HTML encodes all the inputs with the UTF-8 encoding and a UTF-7 encoding attack can be prevented [24].

```
<?php
foreach($_GET as $key => $value){
  echo "The key ".htmlspecialchars($key,ENT_QUOTES,'UTF-8')." has
the value ".htmlspecialchars($value, ENT_QUOTES,'UTF-8')." <br>";
}
?>
```

Fig. 5. Secure coding by escaping data on output.

B. Proper use of PHP Printing Functions

Developers should make proper use of PHP printing function. That is, if they want to capture the output of "print_r()" or "var_export()" functions, they need to escape them correctly. As shown in paragraph 3.1, the exclusive use of htmlspecialchars() is not adequate. Additionally, the return parameter of "print_r()" or "var_export()" functions should be set to "True", as shown in figure 6. In this case, this action will prevent the execution of the XSS attack vector, as the

output of “print_r()” and “var_export()” functions will be stored into a variable and hence it will be escaped through the htmlspecialchars() PHP function. Otherwise, the output of those functions will be printed unescaped, resulting in possible execution of the inserted Javascript code, since the output is not passed to the htmlspecialchars().

```
<?php
  //-------------------------------------------------
  // Case 01 - The "print_r()" function.
  //-------------------------------------------------
  echo '<li><b>The "print_r()" function response: </b><br></li>';
  $get = print_r($_GET['x'],True);
  echo htmlspecialchars($get, ENT_QUOTES, 'UTF-8');

  //-------------------------------------------------
  // Case 02 - The "var_export()" function.
  //-------------------------------------------------
  echo '<br><br><b> <li>The "var_export()" function response:
</b><br></li>';
  $get = var_export($_GET['x'],True);
  echo htmlspecialchars($get, ENT_QUOTES, 'UTF-8');
?>
```

Fig. 6. Secure coding by returning True in print_r() and var_export() functions in combination with htmlspecialchars()

C. *Patching XSS Auditor*

To patch the XSS Auditor, we obtained the WebKitGTK+ source code, from the official repository [7], which a port of the web rendering engine WebKit to the GTK+ 3 platform. By the time we performed the patching, the version of WebKitGTK+ was Revision 172889. The patching code has been inserted in a specific file of the XSS Auditor named XSSAuditor.cpp [6]. In essence, our patching code performs several additional security checks to detect PHP Array and Array-like Injection attacks. After our patches, we repeated our experiments and we observed that the WebKit engine was not vulnerable anymore to the identified attacks. The patching code can be found in [25]. Finally, we have committed the patches to the official repository of WebKit on GitHub [7].

V. CONCLUSIONS

In this paper, we focused on the mistakes that PHP web application developers make, primarily on managing variables and unwittingly help the attackers to bypass the XSS Auditor. We presented three real-world examples of badly written PHP code and how an attacker can construct an attack vector to perform an XSS attack. To defend against the identified attacks, we provided secure coding practices for PHP developers. Finally, we showed how we managed to patch the XSS Auditor and enhance its security.

REFERENCES

[1] Netcraft, http://news.netcraft.com/archives/2013/01/31/php-just-grows-grows.html
[2] National Vulnerability Database (NVD), http://web.nvd.nist.gov.
[3] PHP-related vulnerabilities on the National Vulnerability Database, http://www.coelho.net/php_cve.html
[4] OWASP Top Ten 2013, https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013
[5] OWASP, Cross-site-Scripting(XSS), https://www.owasp.org/index.php/Cross-site_Scripting_XSS
[6] XSS Auditor, https://github.com/WebKit/webkit/blob/master/Source/WebCore/html/parser/XSSAuditor.cpp
[7] WebKit, “The WebKit Open Source Project”, https://github.com/WebKit/webkit
[8] David Ross, “IE 8 XSS Filter Architecture/Implementation”, http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx
[9] Noscript, http://www.noscript.net.
[10] Daniel Bates, Adam Barth, and Collin Jackson, “Regular expressions considered harmful in client-side xss filters”, Proceedings of the 19th international conference on World wide web (www 2010), USA.
[11] sla.ckers.org, Chrome gets XSS filters, http://sla.ckers.org/forum/read.php?13,31377
[12] Nick Nikiforakis, Bypassing Chrome’s Anti-XSS filter, http://blog.securitee.org/?p=37
[13] Chromium, Issue 96616: Security: Google Chrome Anti-XSS filter circumvention, https://code.google.com/p/chromium/issues/detail?id=96616
[14] Issue 114641: XSS Auditor bypass with svg tags, https://code.google.com/p/chromium/issues/detail?id=114641
[15] Issue 114346: XSS Auditor bypass with U+2028/2029, https://code.google.com/p/chromium/issues/detail?id=114346
[16] Sebastian Lekies, Ben Stock, Martin Johns, “A tale of the weaknesses of current client-side XSS filtering”, BlackHat USA 2014.
[17] Keith Makan, “Bypassing NoScript's XSS filters via Error Basd SQLi”, http://blog.k3170makan.com/2012/07/nonoscript-bypassing-noscripts-xss.html
[18] R.T. Waysea's Blog, “Of Trusted And Untrusted Data”, http://rtwaysea.net/blog/blog-2013-10-18-long.html
[19] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius and Edward Z. Yang, mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations, Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS 13), Berlin, Germany.
[20] html5rocks, “How Browsers Work: Behind the scenes of modern web browsers”. http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/
[21] Mozilla, “The Nan Property”, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN
[22] OWASP, “XSS (Cross Site Scripting) Prevention Cheat Sheet”, https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet
[23] PHP, “htmlspecialchars() – Convert special characters to HTML entities”, http://php.net/manual/en/function.htmlspecialchars.php
[24] Chris Shiflett, “Google's XSS Vulnerability”, http://shiflett.org/blog/2005/dec/googles-xss-vulnerability
[25] https://github.com/stasinopoulos/webkit/commit/557d41ba23781cd53dedc4d2e40c5af220e8b966