Regular Expression Injection

Christian Wenz

Hauser Wenz Partnerschaftsgesellschaft
chw at hauser dash wenz dot de

Last update: October 24, 2005

Abtract. In order to combat web application security issues, two main aspects must always be considered: Input must be validated, output must be escaped. A lack of input validation can lead to a dangerous injection attack, most prominently known are SQL Injections or command injections, and more recently XPath injections [1]. This paper presents a new way of attack called RegEx Injections/Regular Expression Injections.

1 Introduction

The term „injection" is quite common when it comes to web application security. Most prominent is „SQL Injection", an approach where an attacker injects SQL code and lets the application execute it. Another well-known attack is command injection, where a web application starts a process and the attacker succeeds in injecting something in the call. Álvarez and Pettrvić describe a taxonomy for web attacks and also mention XPath injection where an attacker uses injection to modify an XPath query [1]. Klein took this approach one step further and showed that also blind, automated XPath injections are possible [2].

Unfortunately, this is not the end of possible injections. Many web technologies provide a special modifier for regular expressen queries, *e*, that allows the execution of code. Whereas this alone is no bad idea and rather rarely in use, if applied wrongly this can introduce a very severe security vulnerability in the web application. Section 2 demonstrates how Regular Expression Injection/RegEx Injection works, section 3 shows counter measures and section 3 concludes this paper.

2 Regular Expression Injection

Regular Expression Injection or RegEx Injection uses the substitution modifier *e* for regular expressions to inject code in a web application. Perl supports this modifier, and other technologies include a PCRE (Perl-compatible regular expressions) library like for instance PHP. If this is used, the substitution term is evaluated. This can be very powerful, since this does not only allow variable names to be used, but also any other commands the technology supports.

An example for Regular Expression Injection is given as follows. Consider a system where user credentials are saved in an XML file like this (a similar example was originally used in Klein´s paper on Blind XPath Injection [2]):

XML file that stores user information

```
<?xml version="1.0"?>
<userdata>
  <user>
    <username>administrator</username>
    <password>Secret</password>
  </user>
  <user>
    <username>John Doe</username>
    <password>SecretAsWell</password>
  </user>
</userdata>
```

Also, consider the following PHP script to change the user password. For some reason, a regular expression is used for that task (file is read in, the old password is replaced by the new one, file is written back). Also, the *e* modifier is used to convert the username in lowercase—in the sample system, user names are case-insensitive so they are stored in lowercase only, saving one call to *strtolower()* or a related function when logging in. The following code demonstrates the codes used for that; mind that line breaks withing regular expressions are used for legibility only.

PHP code to change a user´s password, vulnerable to RegEx Injection

```
<?php
  // input: $username, $password, $newpassword
  // e.g.:
  //   $username = (isset($_POST['username'])) ?
  //               $_POST['username'] : '';
  //   $password = (isset($_POST['password'])) ?
  //               $_POST['password'] : '';
  //   $newpassword = (isset($_POST['newpassword'])) ?
  //                  $_POST['newpassword'] : '';

  $users = file_get_contents('users.xml');
  $users = preg_replace(
    "#(<username\s*>\s*$username\s*</username>\s*
    <password\s*>)$password</password>#e",
    "'\\1'.strtolower('$newpassword').'</password>'",
    $users
  );
  file_put_contents('users.xml', $users);
?>
```

The modifier *e* was used to make sure that *strtolower()* is called, however user-supplied data (in this case, the variable *$newpassword*) is made part of the replacement term of the call to *preg_replace()*. Therefore, the code is vulnerable to

Regular Expression Injection. Consider the following password being sent to the script in *$_POST['newpassword']*:

```
'.system('reboot').'
```

Then, the replacement term turns out as:

```
"'\\1'.strtolower(''.system('reboot').'').'</password>'"
```

So the call to the *system()* function was injected in the application. While this specific attack will have no effect on most systems since the user PHP is running under does not have the rights to call *reboot*, the implications of such a security vulnerability are obvious, since PHP functions and also system calls can be executed if not deactivated in the configuration.

3 Countermeasures

The preceding exploit only worked because the string parameter to *strtolower()* could contain single quotes, ending the string. Therefore, escaping all quotes using the backslash may be a viable solution—PHP offers *addslashes()* for automatically fulfilling this request. On the other hand, this generates an undesired output when the replacement string itself contains backslashes. Therefore, avoiding the *e* modifier for regular expressions is considered as the best and safest solution. PHP offers an alternative way to execute code when *preg_replace()* is called in forms of the function *preg_replace_callback()*.

4 Conclusion

This paper introduced a novel web application security attack called Regular Expression Injection or RegEx Injection. If code is vulnerable to it, the implications are similar to the ones of code injection. Although real-world applications quite rarely user the *e* modifier in Perl-compatible regular expressions, the attack is very dangerours and the author has found one instance of it (with no resemblance to the example presented in this paper) in a web application some time ago.

Since this is a new idea, there are several options where research could continue. For instance, it may be interesting to evaluate more web technologies on their support for the *e* modifier. Also, the error messages when the attack does not work but make the script fail could be analyzed to fingerprint the application. Ultimately, it may be interesting to find out whether Bling RegEx Injection is possible.

References

1. Álvarez, G., Petrovi ć, S.: Encoding a Taxonomy of Web Attacks with Different-Length Vectors. http://arxiv.org/PS_cache/cs/pdf/0210/0210026.pdf
2. Klein, A.: Blind XPath Injection. http://www.packetstormsecurity.org/papers/bypass/Blind_XPath_Injection_20040518.pdf