

Detection of SQL Injection and Cross-site Scripting Attacks

K. K. Mookhey, Nilesh Burghate 2004-03-17

1. Introduction

In the last couple of years, attacks against the Web application layer have required increased attention from security professionals. This is because no matter how strong your firewall rulesets are or how diligent your patching mechanism may be, if your Web application developers haven't followed secure coding practices, attackers will walk right into your systems through port 80. The two main attack techniques that have been used widely are SQL Injection [ref 1] and Cross Site Scripting [ref 2] attacks. SQL Injection refers to the technique of inserting SQL meta-characters and commands into Web-based input fields in order to manipulate the execution of the back-end SQL queries. These are attacks directed primarily against another organization's Web server. Cross Site Scripting attacks work by embedding script tags in URLs and enticing unsuspecting users to click on them, ensuring that the malicious Javascript gets executed on the victim's machine. These attacks leverage the trust between the user and the server and the fact that there is no input/output validation on the server to reject Javascript characters.

This article discusses techniques to detect SQL Injection and Cross Site Scripting (CSS) attacks against your networks. There has been a lot of discussion on these two categories of Web-based attacks about how to carry them out, their impact, and how to prevent these attacks using better coding and design practices. However, there is not enough discussion on how these attacks can be detected. We take the popular open-source IDS Snort [ref 3], and compose regular-expression based rules for detecting these attacks. Incidentally, the default ruleset in Snort does contain signatures for detecting cross-site scripting, but these can be evaded easily. Most of them can be evaded by using the hex-encoded values of strings such as **%3C%73%63%72%69%70%74%3E** instead of **<script>**.

We have written multiple rules for detecting the same attack depending upon the organization's level of paranoia. If you wish to detect each and every possible SQL Injection attack, then you simply need to watch out for any occurrence of SQL meta-characters such as the single-quote, semi-colon or double-dash. Similarly, a paranoid way of checking for CSS attacks would be to simply watch out for the angled brackets that signify an HTML tag. But these signatures may result in a high number of false positives. To avoid this, the signatures can be modified to be made accurate, yet still not yield too many false positives.

Each of these signatures can be used with or without other verbs in a Snort signature using the **pcrc** [ref 4] keyword. These signatures can also be used with a utility like grep to go through the Web-server's logs. But the caveat is that the user input is available in the Web server's logs only if the application uses GET requests. Data about POST requests is not available in the Web server's logs.

2. Regular Expressions for SQL Injection

An important point to keep in mind while choosing your regular expression(s) for detecting SQL Injection attacks is that an attacker can inject SQL into input taken from a form, as well as through the fields of a cookie. Your input validation logic should consider each and every type of input that originates from the user -- be it form fields or cookie information -- as suspect. Also if you discover too many alerts coming in from a signature that looks out for a single-quote or a semi-colon, it just might be that one or more of these characters are valid inputs in cookies created by your Web application. Therefore, you will need to evaluate each of these signatures for your particular Web application.

As mentioned earlier, a trivial regular expression to detect SQL injection attacks is to watch out for SQL specific meta-characters such as the single-quote (') or the double-dash (--). In order to detect these characters and their hex equivalents, the following regular expression may be used:

2.1 Regexp for detection of SQL meta-characters

```
/(\%27)|(\')|(\-\-)|(\%23)|(\#)/ix
```

Explanation:

We first detect either the hex equivalent of the single-quote, the single-quote itself or the presence of the double-dash. These are SQL characters for MS SQL Server and Oracle, which denote the beginning of a comment, and everything that follows is ignored. Additionally, if you're using MySQL, you need to check for presence of the '#' or its hex-equivalent. Note that we do not need to check for the hex-equivalent of the double-dash, because it is not an HTML meta-character and will not be encoded by the browser. Also, if an attacker tries to manually modify the double-dash to its hex value of %2D (using a proxy like Achilles [ref 5]), the SQL Injection attack fails.

The above regular expression would be added into a new Snort rule as follows:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"SQL Injection - Paranoid"; flow:to_server,established;uricontent:".pl"; pcre:"/(\%27)|(\')|(\-\-)|(%23)|(#)/i"; classtype:Web-application-attack; sid:9099; rev:5;)
```

In this case, the **uricontent** keyword has the value ".pl", because in our test environment, the CGI scripts are written in Perl. Depending upon your particular application, this value may be either ".php", or ".asp", or ".jsp", etc. From this point onwards, we do not show the corresponding Snort rule, but instead only the regular expressions that are to be used for creating these rules. From the regular expressions you can easily create more Snort rules.

In the previous regular expression, we detect the double-dash because there may be situations where SQL injection is possible even without the single-quote [ref 6]. Take, for instance, an SQL query which has the **where** clause containing only numeric values. Something like:

```
select value1, value2, num_value3 from database
where num_value3=some_user_supplied_number
```

In this case, the attacker may execute an additional SQL query, by supplying an input like:

```
3; insert values into some_other_table
```

Finally, pcre modifiers 'i' and 'x' are used in order to match without case sensitivity and to ignore whitespaces, respectively.

The above signature could be additionally expanded to detect the occurrence of the semi-colon as well. However, the semi-colon has a tendency to occur as part of normal HTTP traffic. In order to reduce the false positives from this, and also from any normal occurrence of the single-quote and double-dash, the above signature could be modified to first detect the occurrence of the = sign. User input will usually occur as a GET or a POST request, where the input fields will be reflected as:

```
username=some_user_supplied_value&password=some_user_supplied_value
```

Therefore, the SQL injection attempt would result in user input being preceded by a = sign or its hex equivalent.

2.2 Modified regex for detection of SQL meta-characters

```
/((\%3D) | (=)) [^\n]* ((\%27) | (\') | (\-\-\) | (\%3B) | (;)) /i
```

Explanation:

This signature first looks out for the = sign or its hex equivalent (%3D). It then allows for zero or more non-newline characters, and then it checks for the single-quote, the double-dash or the semi-colon.

A typical SQL injection attempt of course revolves around the use of the single quote to manipulate the original query so that it always results in a true value. Most of the examples that discuss this attack use the string **1'or'1'='1**. However, detection of this string can be easily evaded by supplying a value such as **1'or2>1--**. Thus the only part that is constant in this is the initial alphanumeric value, followed by a single-quote, and then followed by the word 'or'. The Boolean logic that comes after this may be varied to an extent where a generic pattern is either very complex or does not cover all the variants. Thus these attacks can be detected to a fair degree of accuracy by using the next regular expression, in section 2.3 below.

2.3 Regex for typical SQL Injection attack

```
/\w*((\%27) | (\'))((\%6F) | o | (\%4F))((\%72) | r | (\%52)) /ix
```

Explanation:

\w* - zero or more alphanumeric or underscore characters

(\%27) | \' - the ubiquitous single-quote or its hex equivalent

(\%6F) | o | (\%4F)) (\%72) | r | (\%52) - the word 'or' with various combinations of its upper and lower case hex equivalents.

The use of the 'union' SQL query is also common in SQL Injection attacks against a variety of databases. If the earlier regular expression that just detects the single-quote or other SQL meta characters results in too many false positives, you could further modify the query to specifically check for the single-quote and the keyword 'union'. This can also be further extended to other SQL keywords such as 'select', 'insert', 'update', 'delete', etc.

2.4 Regex for detecting SQL Injection with the UNION keyword

```
/((\%27)|(\'))union/ix
```

`(\%27)|(\')` - the single-quote and its hex equivalent

union - the keyword union

Similar expressions can be written for other SQL queries such as **>select, insert, update, delete, drop**, and so on.

If, by this stage, the attacker has discovered that the Web application is vulnerable to SQL injection, he will try to exploit it. If he realizes that the back-end database is on an MS SQL server, he will typically try to execute one of the many dangerous stored and extended stored procedures. These procedures start with the letters 'sp' or 'xp' respectively. Typically, he would try to execute the 'xp_cmdshell' extended procedure, which allows the execution of Windows shell commands through the SQL Server. The access rights with which these commands will be executed are those of the account with which SQL Server is running -- usually Local System. Alternatively, he may also try and modify the registry using procedures such as xp_regread, xp_regwrite, etc.

2.5 Regex for detecting SQL Injection attacks on a MS SQL Server

```
/exec(\s|\+)+(s|x)p\w+/ix
```

Explanation:

exec - the keyword required to run the stored or extended procedure

`(\s|\+)+` - one or more whitespaces or their HTTP encoded equivalents

`(s|x)p` - the letters 'sp' or 'xp' to identify stored or extended procedures respectively

`\w+` - one or more alphanumeric or underscore characters to complete the name of the procedure

3. Regular Expressions for Cross Site Scripting (CSS)

When launching a cross-site scripting attack, or testing a Website's vulnerability to it, the attacker may first issue a simple HTML formatting tag such as `` for bold, `<i>` for italic or `<u>` for underline. Alternatively, he may try a trivial script tag such as `<script>alert("OK")</script>`. This is likely because most of the printed and online literature on CSS use this script as

an example for determining if a site is vulnerable to CSS. These attempts can be trivially detected. However, the advanced attacker may attempt to camouflage the entire string by entering its Hex equivalents. So the `<script>` tag would appear as `%3C%73%63%72%69%70%74%3E`. On the other hand, the attacker may actually use a Web Application Proxy like Achilles and reverse the browser's automatic conversion of special characters such as `<` to `%3C` and `>` to `%3E`. So the attack URL will contain the angled brackets instead of their hex equivalents as would otherwise normally occur.

The following regular expression checks for attacks that may contain HTML opening tags and closing tags `<>` with any text inside. It will catch attempts to use `` or `<u>` or `<script>`. The regex is case-insensitive. We also need to check for the presence of angled brackets, as well as their hex equivalents, or `(%3C|<)`. To detect the hex conversion of the entire string, we must check for the presence of numbers as well as the `%` sign in the user input, in other words, the use of `[a-z0-9%]`. This may sometimes result in false-positives, but most of the time will detect the actual attack.

3.1 Regex for simple CSS attack

```
/((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)/ix
```

Explanation:

`((\%3C)|<)` - check for opening angle bracket or hex equivalent

`((\%2F)|\/)*` - the forward slash for a closing tag or its hex equivalent

`[a-z0-9\%]+` - check for alphanumeric string inside the tag, or hex representation of these

`((\%3E)|>)` - check for closing angle bracket or hex equivalent

Snort signature:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"NII
Cross-site scripting attempt"; flow:to_server,established; pcre:"/((\
%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)/i"; classtype:Web-
application-attack; sid:9000; rev:5;)
```

Cross-site scripting can also be accomplished by using the `` technique. The existing default snort signature can be easily evaded. The one supplied in section 3.2 will be much tougher to evade.

3.2 Regex for "<img src" CSS attack

```
/((\%3C)|<)(\%69)|i|(\%49))((\%6D)|m|(\%4D))((\%67)|g|(\%47))[\^\\n]
+(\%3E)|>)/I
```

Explanation:

(\%3C)|<) opening angled bracket or hex equivalent

(\%69)|i|(\%49))((\%6D)|m|(\%4D))((\%67)|g|(\%47)) the letters 'img' in varying combinations of ASCII, or upper or lower case hex equivalents

[\^\\n]+ any character other than a new line following the <img

(\%3E)|>) closing angled bracket or hex equivalent

3.3 Paranoid regex for CSS attacks

```
/((\%3C)|<)[\^\\n]+((\%3E)|>)/I
```

Explanation:

This signature simply looks for the opening HTML tag, and its hex equivalent, followed by one or more characters other than the newline, and then followed by the closing tag or its hex equivalent. This may end up giving a few false positives depending upon how your Web application and Web server are structured, but it is guaranteed to catch anything that even remotely resembles a cross-site scripting attack.

For an excellent reference on types of cross-site scripting attacks that will evade filters, see the Bugtraq posting <http://www.securityfocus.com/archive/1/272037>. However, note that the last of the cross-site scripting signatures, which is the paranoid signature, will detect all these attacks.

4. Conclusion

In this article, we've presented different types of regular expression signatures that can be used to detect SQL Injection and Cross Site Scripting attacks. Some of the signatures are simple yet paranoid, in that they will raise an alert even if there is a hint of an attack. But there is also the possibility that these paranoid signatures may result in false positives. To take care of this, we've then modified the simple signatures with additional pattern checks so that they are more accurate. We recommend that these signatures be taken as a starting point for tuning your IDS

or log analysis methods, in the detection of these Web application layer attacks. After a few modifications, and after taking into account the non-malicious traffic that occurs as part of your normal Web transactions, you should be able to accurately detect these attacks.

References

1. SQL Injection <http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>
2. Cross Site Scripting FAQ <http://www.cgisecurity.com/articles/xss-faq.shtml>
3. The Snort IDS <http://www.snort.org>
4. Perl-compatible regular expressions (pcre) <http://www.pcre.org>
5. Web application proxy, Achilles <http://achilles.mavensecurity.com>
3. Advanced SQL Injection http://www.nextgenss.com/papers/advanced_sql_injection.pdf
7. Secure Programming HOWTO, David Wheeler www.dwheeler.com
8. Threats and Countermeasures, MSDN, Microsoft <http://msdn.microsoft.com>

About the authors

K. K. Mookhey is Founder & CTO of Network Intelligence India Pvt. Ltd. (www.nii.co.in), which provides information security consulting services including security audits, penetration testing, security design, application audits, and security training. He has written a number of articles and whitepapers on information security, and is also responsible for NII's research initiatives.

Nilesh Burghate is an Information Assurance Consultant with NII, and his interests include penetration testing, IDS signature writing, intrusion analysis and detection and forensics. The results from his team's research efforts provide direct input to NII's Security Alerting service.

[Privacy Statement](#)

Copyright 2006, SecurityFocus