SpiderLabs Anterior

Official Blog of Trustwave's SpiderLabs - SpiderLabs is an elite team of ethical hackers, investigators and researchers at Trustwave advancing the security capabilities of leading businesses and organizations throughout the world.

23 August 2011

ModSecurity Advanced Topic of the Week: (Updated) Exception Handling

UPDATE - since this original post, we added new exception handling capabilities to v2.6.0 which are a tremendous help for adding in custom exceptions. See the section below on Updating the Target Lists.

This post is long overdue. I will cover the current state of exception handling options within both ModSecurity and the OWASP Core Rule Set (CRS).

Exception Handling Methodologies
Before continuing with this blog post, I highly recommend that you review the blog post describing the Traditional vs. Anomaly Scoring Detection Modes. Your detection operating mode will directly impact your exception handling options.

False Positives and WAFs

It is inevitable; you will run into some False Positives (FP) when using web application firewalls. This is not something that is unique to ModSecurity. All web application firewalls will generate some level of false positives, especially when you first deploy them or when your application changes. Continuous application profiling, where the WAF learns expected behavior helps to reduce FPs however negative security model (blacklist) rule sets will always generate some level of FP as they have no idea what input is valid. The following information will help to guide you through the process of identifying, fixing, implementing and testing new exceptions to address false positives within the OWASP ModSecurity CRS.

False Positives In New Environments

False Positives happen with ModSecurity + CRS mainly as a by product of the fact that the rules are ***generic*** in nature. The plug-n-play nature of the CRS what makes it great, as you will get protection for just about any environment however there will be some level of FPs. This ends up being the old "80/20 rule" of security where you will instantly get coverage for about 80% of the problem. The issue then moves towards that remaining 20%. This is where the CRS runs into both false positives and false negatives as there is no way to know exactly what web application is going to be run behind it. That is why the CRS is geared towards blocking the known bad stuff and forcing some HTTP compliance. This catches the vast majority of attacks.

Use DetectionOnly Mode Initially

Any new installation should initially use the "SecRuleEngine DetectionOnly" directive. Review/edit the modsecurity_crs_10_config.conf file or your own local config to verify this setting. After running ModSecurity in a detection only mode for a while review the events and decide if any modification to the rule set should be made before moving to protection mode (SecRuleEngine On and using a Disruptive action in the SecDefaultAction directive).

Review the Rule ID Documentation Pages on the OWASP Project Site

ModSecurity's rules are open source which this allows the user to see exactly what the rule is matching on and also allows you to create your own rules. With closed-source rules, you can not verify what it is looking for so you really have no other option but to remove the offending rule. With the CRS, you can review Rule ID documentation pages on the OWASP site. Each rule has (or will have - as we are just starting this documentation effort) its own documentation page. Here is an example - http://www.owasp.org/index.php/ModSecurity_CRS_RuleID-960911. These pages provide information such as Rule Logic, RegEx Analysis, False Positive info, etc... These pages should help you to better understand the rationale for each rule, what they are looking for and how they are detecting the issues. This information will help you to better decide how to proceed.

Analyzing/Understanding ModSecurity Events

In order to verify if you indeed have a false positive, you need to review your logs. Most people initially start with the Apache error_log file as this contains the short, 1 line ModSecurity Alert Message. It is important that you fully understand the ModSecurity Audit Message Format.
Let's supposed that your site received this GET request -

```
http://yoursite/message_board.php?comment=i%20need%20to%20select%20a%20new%20user%20for%20my%20fantasy%20football%20team%20-%20who%
```

This request would trigger the following ModSecurity CRS SQL Injection Attack alert:

```
[Tue Nov 16 11:44:11 2010] [error] [client ::1] ModSecurity: Warning. Pattern match "\\bselect\\b.{0,40}\\buser\\b" at ARGS:comment
[file "/usr/local/apache/conf/modsec_current/base_rules/modsecurity_crs_41_sql_injection_attacks.conf"] [line "67"]
[id "959514"] [rev "2.0.9"] [msg "Blind SQL Injection Attack"]
[data "select a new user"] [severity "CRITICAL"] [tag "WEB_ATTACK/SQL_INJECTION"] [tag "WASCTC/WASC-19"]
[tag "OWASP_TOP_10/A1"] [tag "OWASP_AppSensor/CIE1"] [tag "PCI/6.5.2"] [hostname "localhost"] [uri "/message_board.php"] [unique_id
```

Let's review this audit log entry so that you understand each piece of data:

Alert Action Description

The first part of the engine message tells you whether ModSecurity acted to interrupt transaction or rule processing:

```
ModSecurity: Warning.
```

This means that this alert was only a warning. If it had said "***ModSecurity: Access denied...***" then it would indicate that the request was blocked.

Alert Justification Description

The second part of the engine message explains ***why*** the alert was generated:

```
ModSecurity: Warning. Pattern match "\\bselect\\b.{0,40}\\buser\\b"
```

The text ***Pattern match*** - means that there was a match for the @rx operator. The last part of this section tells you where the operator matched occurred:

```
ModSecurity: Warning. Pattern match "\\bselect\\b.{0,40}\\buser\\b" at ARGS:comment
```

In this case, the operator found a match in the ***comment*** parameter payload. This data is useful if you decide that you need to do an exception for a specify parameter value.

Meta-Data

The metadata fields are always placed at the end of the alert entry. Each metadata field is a text fragment that consists of an open bracket followed by the metadata field name, followed by the value and the closing bracket.

```
[Tue Nov 16 11:44:11 2010] [error] [client ::1] ModSecurity: Warning. Pattern match "\\bselect\\b.{0,40}\\buser\\b" at ARGS:comment
[file "/usr/local/apache/conf/modsec_current/base_rules/modsecurity_crs_41_sql_injection_attacks.conf"] [line "67"]
[id "959514"] [rev "2.0.9"] [msg "Blind SQL Injection Attack"]
[data "select a new user"] [severity "CRITICAL"] [tag "WEB_ATTACK/SQL_INJECTION"] [tag "WASCTC/WASC-19"]
[tag "OWASP_TOP_10/A1"] [tag "OWASP_AppSensor/CIE1"] [tag "PCI/6.5.2"] [hostname "localhost"]
[uri "/message_board.php"] [unique_id "TOK028CoC2oAAAZ8FkkAAAAA"]
```

The following metadata fields are currently used in this alert and are useful for implementing exceptions:

1. `id` - Unique rule ID, as specified by the `id` action.
2. `msg` - Human-readable message, as specified by the `msg` action.
3. `uri` - Request URI.
4. `data` - contains transaction data fragment, as specified by the `logdata` action.

By analyzing the data within this alert, especially the matched logdata, you should have enough information to make an estimation as to whether or not the alert might be a false positive. Sometimes, however, you can only confirm whether or not the alert is a false positive by reviewing the entire transactional payloads. This means that you need to review the audit_log file.

Audit Log
The audit log data is tremendously useful for providing full request payloads which will allow you to better confirm any FP matches. In the example above, the comment field data is in the query_string variable which would be present in standard Apache access_log data. If, however, this had been a POST request and the comment field was in the POST_PAYLOAD then you would need to review the audit log to see the matched data from the alert. Here is an example audit_log entry for this request:

```
--3b372d22-A--
[29/Nov/2010:13:19:04 --0500] TPPumMCoC2oAAAdQDNgAAAAB ::1 51733 ::1 80
--3b372d22-B--
GET /message_board.php?comment=i%20need%20to%20select%20a%20new%20user%20for%20my%20fantasy%20football%20team%20-%20who%20should%20
Host: yourhost
Connection: keep-alive
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_5; en-US) AppleWebKit/534.7 (KHTML, like Gecko) Chrome/7.0.517.44 Safari
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3


--3b372d22-F--
HTTP/1.1 403 Forbidden
Content-Length: 219
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1


--3b372d22-H--
Message: Warning. Pattern match "\bselect\b.{0,40}\buser\b" at ARGS:comment. [file "/usr/local/apache/conf/modsec_current/base_rule
Message: Access denied with code 403 (phase 2). Pattern match "(.*)" at TX:0. [file "/usr/local/apache/conf/modsec_current/base_rul
Message: Warning. Operator GE matched 5 at TX:inbound_anomaly_score. [file "/usr/local/apache/conf/modsec_current/base_rules/modsec
Stopwatch: 1291054744363338 123923 (8315 122194 -)
Producer: ModSecurity for Apache/2.5.13dev3 (http://www.modsecurity.org/); core ruleset/2.0.9.
Server: Apache/2.2.17 (Unix) mod_ssl/2.2.12 OpenSSL/0.9.8l DAV/2


--3b372d22-Z--
```

After reviewing the audit_log data, it is pretty evident that this alert is a FP. The rule improperly triggered a Blind SQL Injection Attack alert in the comment field. This is a common occurrence in free-form text fields where clients can write anything that they want.

Debug Log
The last place to look, and actually the best source of information, is the modsec_debug.log file. This file can show everything that ModSecurity is doing, especially if you turn up the **`SecDebugLogLevel`** to 9. Keep in mind, however, that increasing the verboseness of the debug log does impact performance. While increasing the verboseness for all traffic is usually not feasible, what you can do is to create a new rule that uses the “ctl” action to turn up the debugloglevel selectively. For instance, if you identify a False Positive from only one specific user, you could add in a rule such as this (adjusting the proper source IP address) to a local custom rules file such as modsecurity_crs_15_customrules.conf:

```
SecRule REMOTE_ADDR "^192\.168\.10\.69$" phase:1,log,pass,ctl:debugLogLevel=9
```

This will set the debugLogLevel to 9 only for requests coming from that specific source IP address. You want to make sure that these rules run ***before*** the existing CRS rules. Perhaps that still generates a bit too much traffic. You could tighten this down a bit to increase the logging only for the specific file or argument that is causing the false positive:

```
SecRule REQUEST_URI "^/path/to/script.pl$" phase:1,log,pass,ctl:debugLogLevel=9

or

SecRule ARGS:variablename “something” phase:1,pass,ctl:debugLogLevel=9
```

Now that you have verbose information in the debug log file, you can review it to ensure that you understand what portion of the request was being inspected when the specific rule trigger and you can also view the payload after all of the transformation functions have been applied.

Don't be too hasty to remove a rule

Just because a particular rule is generating a false positive on your site does not mean that you should immediately jump to removing the rule entirely. Remember, these rules were created for a reason. They are intended to block a known attack. By removing this rule completely, you might expose your website to the very attack that the rule was created for. This would be the dreaded False Negative. We will present many different methods of externally adjusting the rules to address the FP.

Try to avoid altering the Core Rules
In general, it is recommended that you try to limit your alteration of the Core Rule Set files as much as possible. The more you alter the individual, existing rules themselves, the less likely it will be that you will want to upgrade to the newer releases since you would have to recreate your customizations. What we recommend is that you try to contain your changes to your own custom rules file(s) that are particular to your site. This is where you would want to add new signatures and to also create rules to exclude False Positives from the normal Core Rules files.

Fixing False Positives - Traditional Mode
Adding new white-listing rules
If you need to add new white-listing rules so that you can, for instance, allow a specific client IP address to pass through all of the ModSecurity rules you should place this type of rule after the modsecurity_crs_10_config.conf file but BEFORE the other Core Rules. This is accomplished by creating a new rule file called – modsecurity_crs_15_customrules.conf and place it in the same directory as the other Core Rules. This is assuming you are using the Apache Include directive to call up the Core Rules like this –

```
<IfModule security2_module>

Include conf/rules/*.conf

</IfModule>
```

By naming your file with the "_15_" string in it, it will be called up just after the config file. This will ensure that your new white-list rule will be executed early and you can then use such actions as allow and ctl:ruleEngine=Off to allow the request through the remainder of the rules.

```
SecRule REMOTE_ADDR "^192\.168\.10\.69$" phase:1,log,allow:request
```

This rule would allow the request to bypass phase:1 and phase:2 rules and then pick up processing again in phase:3. This type of rules is useful when you want to allow an authorized vulnerability scanning source to scan your web app but you don't want to trigger alerts on inbound requests.

(New) Explicitly Updating the Target List
In ModSecurity v2.6, we introduced the SecRuleUpdateTargetById directive (and it's ctl equivalent) which make creating local exceptions much easier. This is useful for implementing exceptions where you want to externally update a target list to exclude inspection of specific variable(s). Let's say you have the following rule ID 958895:

```
SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/* "[\;\|\`]\W*?\bmail\b" \
     "phase:2,rev:'2.1.1',capture,t:none,t:htmlEntityDecode,t:compressWhitespace,t:lowercase,ctl:auditLogParts=+E,block,msg:'System
{tx.0}"
```

Let's also say that you are running into false positives for a parameter called "email" and you would like to remove this location from inspection, you could add the following directive to a local custom rules file that comes ***after*** the existing CRS rules. Call this new file something like – **modsecurity_crs_60_customrules.conf**. Just make sure that number in the filename is higher than any other rules file so it is read last.

```
SecRuleUpdateTargetById 958895 !ARGS:email
```

The effective resulting rule in the previous example will append the target to the end of the variable list as follows:

```
SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/*|!ARGS:email "[\;\|\`]\W*?\bmail\b" \
     "phase:2,rev:'2.1.1',capture,t:none,t:htmlEntityDecode,t:compressWhitespace,t:lowercase,ctl:auditLogParts=+E,block,msg:'System
{tx.0}""
```

(New) Conditionally Updating the Target List
While the previous example does work, usually you only need to remove a parameter from inspection for a specific URL resource. In this case, you can also do the same by using the ctl action. This is useful if you want to only update the targets for a particular URL

```
SecRule REQUEST_FILENAME "@streq /path/to/file.php" \
"phase:1,t:none,nolog,pass,ctl:ruleUpdateTargetById=958895;!ARGS:email"
```

Explicitly Removing Rules
If for some reason you can not update the Target list as outlined previously and you want to instead remove a particular SecRule ***explicitly***, you can use the SecRuleRemoveByID directive like this:

```
SecRuleRemoveById 959514
```

This rule needs to go in a local custom rules file that comes ***after*** the existing CRS rules. Call this new file something like – **modsecurity_crs_60_customrules.conf**. Just make sure that number in the filename is higher than any other rules file so it is read last. SecRuleRemoveById needs to be defined after the existing rule it is removing. Similarly, you can use the SecRuleRemoveByMsg directive like this:

```
SecRuleRemoveByMsg "Blind SQL Injection Attack"
```

This directive would, however, remove all SecRules that have that particular message and not just the one that is causing the FP.

Conditionally Removing Rules
If want to remove a particular SecRule explicitly, you can use the ctl:ruleRemoveById action. This method has the benefit of being more precise over exactly when to remove a rule vs. the SecRuleRemoveById directive which removes the rule entirely regardless of the request data. If we wanted to conditionally remove ruleID 959514 only when the URL was "message_board.php" we could do it like:

```
SecRule REQUEST_FILENAME "@streq /message_board.php" "phase:1,t:none,nolog,pass,ctl:ruleRemoveById=959514"
```

This new exception rule should go in the **modsecurity_crs_15_customrules.conf**. While this approach does help with regards to balancing FPs and FNs, it still is not as good as it could be.

Updating the Rule ID Action
If you are running the OWASP ModSecurity CRS in Traditional/blocking mode, then the false positive may be causing you problems as it could be blocking a non-malicious user. Another option that you have is to externally update the action list for the rule ID that is causing the FP. For instance, I could add the following directive to my modsecurity_crs_60_customrules.conf file:

```
SecRuleUpdateActionById 959514 "pass"
```

This would update the action for rule ID 959514 so that it would not deny the request and would alert only.

Updating the Rule ID Target List
A better approach to fixing the FP condition would be to exclude inspection of a specific ARGS payload on a specific page. This could be accomplished by copying the existing CRS rule that was causing the FP (rule ID 959514 in this case) and then copying it to the **modsecurity_crs_15_customrules.conf** file. Once you have placed it within the file, you will then need to update it as follows:

```
SecRule REQUEST_FILENAME "@streq /message_board.php" "chain,phase:2,rev:'2.0.9',t:none,t:urlDecodeUni,t:htmlEntityDecode,t:lowercas
ctl:ruleRemoveById=959514,ctl:auditLogParts=+E,block,msg:'Blind SQL Injection Attack',id:'100',tag:'WEB_ATTACK/SQL_INJECTION',tag:'
    SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/*|!ARGS:comment "\bselect\b.{0,40}\buser\b" \
         "capture,t:none,t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,t:replaceComments,t:compressWhiteSpace,setvar:'tx.msg=%{rule
setvar:tx.sql_injection_score=+%{tx.critical_anomaly_score},setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},
setvar:tx.%{rule.id}-WEB_ATTACK/SQL_INJECTION-%{matched_var_name}=%{tx.0}"
```

The updated rule has done the following:

1. We created a chained rule that is first inspecting the requested filename (/message_board.php) as this is the resource that is triggering the FP. For all other URLs, the normal CRS rule id 959514 will execute.
2. If this matches, we are then using the ctl action to disable the existing CRS rule (959514) that is causing the FP.
3. We are assigning a new rule id to our custom rule (id:'100')
4. In the new TARGET listing - we are excluding inspection of the ARGS:comment variable.

While this approach works, it is a bit kludgy… A better approach, instead of having to copy/paste/edit rules, would be to have a capability to externally update the TARGET listing. This would be similar in concept to the SecRuleUpdateActionById directive except for TARGET variables. **We have an existing JIRA ticket which will implement this type of functionality in the next version of ModSecurity.** Once this is implemented, you would be able to easily create an exception by adding the following rule to your custom rule file:

```
SecRule REQUEST_FILENAME "@streq /message_board.php" "phase:2,t:none,nolog,pass,ctl:ruleUpdateTargetsById=959514:!ARGS:comment"
```

This external approach is preferable vs. copy/paste/editing as this would allow for easier updating of the CRS package. For instance, if a new CRS version updated the operator data (perhaps with a new regular expression) the copy/paste/edit approach would need to be redone.

Fixing False Positives - Anomaly Scoring Mode

Increasing the Anomaly Scoring Blocking Threshold

One of our main motivations with the new CRS anomaly scoring mode was to make it easier for users to handle exceptions. Instead of having to edit rules, you now have an option to simply increase the anomaly scoring blocking threshold TX variable data within the modsecurity_crs_10_config.conf file:

```
SecAction "phase:1,t:none,nolog,pass,setvar:tx.inbound_anomaly_score_level=5"
```

This default setting of 5 means that if there is 1 rule match of a critical severity, then the request would be blocked. One advantage of using anomaly scoring mode, is that you can have an increased confidence in blocking the higher the anomaly score is. So, if you were to simply increase this setting to a value of 10, then this would mean that at least 2 critical severity rules would have to match and this would be less likely to be a FP. ***The bottom line is that the higher the anomaly score, the more certain you can be that the request is malicious and you can safely block it***.

Anomaly Scoring Exceptions

If you did not want to increase the anomaly scoring blocking threshold, you could implement a local exception. The concept is that you let the CRS detection rules process normally and then you implement your exceptions within the **modsecurity_crs_48_local_exceptions.conf** file. This file is the proper place to implement your exceptions as it will be called up ***after*** the normal inbound detection rules and ***before*** the anomaly scoring evaluation and blocking in the **modsecurity_crs_49_inbound_blocking.conf** file.

Let's look again at the CRS rule ID 959514, which was causing the FP:

```
SecRule REQUEST_FILENAME|ARGS_NAMES|ARGS|XML:/* "\bselect\b.{0,40}\buser\b" \
     "phase:2,rev:'2.0.9',capture,t:none,t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,t:replaceComments,t:compressWhiteSpace,
ctl:auditLogParts=+E,block,msg:'Blind SQL Injection Attack',id:'959514',tag:'WEB_ATTACK/SQL_INJECTION',tag:'WASCTC/WASC-19',
tag:'OWASP_TOP_10/A1',tag:'OWASP_AppSensor/CIE1',tag:'PCI/6.5.2',logdata:'%{TX.0}',severity:'2',setvar:'tx.msg=%{rule.msg}',
setvar:tx.sql_injection_score=+%{tx.critical_anomaly_score},setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},
setvar:tx.%{rule.id}-WEB_ATTACK/SQL_INJECTION-%{matched_var_name}=%{tx.0}"
```

The bolded setvar action is the key piece of data to understand. If a rule matches, then we initiate a TX variable that will contain meta-data about the match:

1. `tx.%{rule.id}` - uses macro expansion to capture the rule ID value data and saves it in the TX variable name.
2. `WEB_ATTACK/SQL_INJECTION` - captures the attack category data and saves it in the TX variable name.
3. `%{matched_var_name}` - captures the variable location of the rule match and saves it in the TX variable name.
4. `%{tx.0}` - captures the variable payload data that matched the operator value and saves it in the TX variable value.

If we look at the debug log data when this rule is processing our sample request, we see the following:

```
Executing operator "rx" with param "\\bselect\\b.{0,40}\\buser\\b" against ARGS:comment.
Target value: "i need to select a new user for my fantasy football team - who should i pick"
Added regex subexpression to TX.0: select a new user
Operator completed in 24 usec.
Ctl: Set auditLogParts to ABIFHZE.
Setting variable: tx.msg=%{rule.msg}
Resolved macro %{rule.msg} to: Blind SQL Injection Attack
Set variable "tx.msg" to "Blind SQL Injection Attack".
Setting variable: tx.sql_injection_score=+%{tx.critical_anomaly_score}
Recorded original collection variable: tx.sql_injection_score = "0"
Resolved macro %{tx.critical_anomaly_score} to: 5
Relative change: sql_injection_score=0+5
Set variable "tx.sql_injection_score" to "5".
Setting variable: tx.anomaly_score=+%{tx.critical_anomaly_score}
Recorded original collection variable: tx.anomaly_score = "0"
Resolved macro %{tx.critical_anomaly_score} to: 5
Relative change: anomaly_score=0+5
Set variable "tx.anomaly_score" to "5".
Setting variable: tx.%{rule.id}-WEB_ATTACK/SQL_INJECTION-%{matched_var_name}=%{tx.0}
Resolved macro %{rule.id} to: 959514
Resolved macro %{matched_var_name} to: ARGS:comment
Resolved macro %{tx.0} to: select a new user
Set variable "tx.959514-WEB_ATTACK/SQL_INJECTION-ARGS:comment" to "select a new user".
Resolved macro %{TX.0} to: select a new user
Warning. Pattern match "\bselect\b.{0,40}\buser\b" at ARGS:comment. [file "/usr/local/apache/conf/modsec_current/base_rules/modsecu
```

The final bolded entry shows you the final TX variable data that is now at our disposal. Keep in mind that the individual pieces of data captured in the TX data is also available from within the standard error_log entry as outlined in the previous sections.

Although this rule has triggered and generated a FP, at this point, the only problem is that it has erroneously increased the overall anomaly score by 5 points. With this saved TX data, however, we could easily add the following exception rule to the modsecurity_crs_48_local_exceptions.conf file and re-adjust the anomaly score:

```
SecRule REQUEST_FILENAME "@streq /message_board.php" "chain,phase:2,t:none,log,pass,msg:'Adjusting FP Score'"
    SecRule &TX:959514-WEB_ATTACK/SQL_INJECTION-ARGS:comment "@ge 1" "setvar:tx.anomaly_score=-5"
```

This chained rule verifies the URL resource and then it checks to see if there was a previous rule match from rule id 959514 in the comment parameter value. If so, then it will re-adjust the anomaly score by subtracting 5 points. The end result of this exception is that the request would not be blocked by the anomaly scoring evaluation rules in the **modsecurity_crs_49_inbound_blocking.conf** file due to the initial FP match.

Easy Implementation of new Core Rules

With this type of methodology, you can create custom exclusions and fix false positives and it also allows for easy updating of the Core Rules themselves. What we don't want to have happen is that current ModSecurity users have altered the Core Rules files extensively for their environment that they do not want to upgrade when new Core Rule releases are available for fear of having to re-implement all of their custom configs. With this scenario, you can download new Core Rules versions as they are released and then just copy over your new ModSecurity custom rule files and you are ready to go!

Posted by Ryan Barnett on 23 August 2011 at 10:14 in ModSecurity, ModSecurity Rules | Permalink ShareThis

Comments

Verify your Comment

Previewing your Comment

Posted by: |

This is only a preview. Your comment has not yet been posted.

Post Edit

Your comment could not be posted. Error type:

Your comment has been posted. Post another comment

The letters and numbers you entered did not match the image. Please try again.

As a final step before posting your comment, enter the letters and numbers you see in the image below. This prevents automated programs from posting comments.

Having trouble reading this image? View an alternate.

Continue