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.

Home
Archives
Subscribe
About SpiderLabs
Careers at Trustwave
17 November 2010

Advanced Topic of the Week: Traditional vs. Anomaly Scoring Detection Modes

In the latest SVN trunk version of the CRS (2.0.9), we have implemented the capability for users to easily toggle between ***Traditional*** or ***Anomaly Scoring*** detection modes. This will most likely come as very welcomed enhancement for many users. With the initial CRS v2.0, I feel that we jumped the gun a bit and in reality forced end users into using an Anomaly Scoring detection mode. In hindsight, this was not the right thing to do. The CRS should not force users into using any one specific mode of operation. So, we went back to the proverbial drawing-board and implemented a number of updates which now allow a user to more easy switch between detection modes of operation.

Once you have downloaded and unpacked the CRS archive, you should review/update the new **`modsecurity_crs_10_config.conf.example`** file. This is the central configuration file which will allow you to control how the CRS will work. In this file, you can control the following related CRS items:

- Mode of Detection – Traditional vs. Anomaly Scoring
- Anomaly Scoring Severity Levels
- Anomaly Scoring Threshold Levels (Blocking)
- Enable/Disable Blocking
- Choose where to log events (Apache error_log and/or ModSecurity's audit log)

In order to facilitate the operating mode change capability, we had to make some changes to the rules. Specifically, most rules now use the generic **`block`** action instead of specifying any exact action to take. This change makes it easy for the user to adjust settings in the SecDefaultAction and these will become inherited by the SecRules. This is a good approach to use for a 3rd party set of rules as our goal is ***detection of issues*** and not telling the user ***how to react*** to it. We also removed the logging actions from the rules so that the user can now also control exactly where they want the rules to log alerts to.

Please review this entire blog post so that you will have a better understanding of your operating mode options and you can make an informed decision about your selection.

CRS Operating Modes

Traditional Detection Mode - Self-Contained Rules Concept
Traditional Detection Mode (or IDS/IPS mode) is the new default operating mode. This is the most basic operating mode where all of the rules are "self-contained." Just like HTTP itself, the individual rules are stateless. This means that no intelligence is shared between rules and each rule has no information about any previous rule matches. It only uses its current, single rule logic for detection. In this mode, if a rule triggers, it will execute any disruptive/logging actions specified on the current rule.
Configuring Traditional Mode
If you want to run the CRS in Traditional mode, you can do this easily by verifying the SecDefaultAction line in the modsecurity_crs_10_config.conf file uses a disruptive action such as deny:

```
#
# -=[ Mode of Operation ]=-
#
# You can now choose how you want to run the modsecurity rules -
#
#       Anomaly Scoring vs. Traditional
#
# Each detection rule uses the "block" action which will inherit the SecDefaultAction
# specified below.  Your settings here will determine which mode of operation you use.
#
# Traditional mode is the current default setting and it uses "deny" (you can set any
# disruptive action you wish)
#
# If you want to run the rules in Anomaly Scoring mode (where blocking is delayed until the
# end of the request phase and rules contribute to an anomaly score) then set the
# SecDefaultAction to "pass"
#
# You can also decide how you want to handle logging actions.  You have three options -
#
#       - To log to both the Apache error_log and ModSecurity audit_log file use - log
#       - To log *only* to the ModSecurity audit_log file use - nolog,auditlog
#       - To log *only* to the Apache error_log file use - log,noauditlog
#
SecDefaultAction "phase:2,deny,log"
```

This setting mimics the previous CRS detection mode. When a CRS rule matches, it will be denied and then the alert data will be logged to both the Apache error_log file and ModSecurity Audit log file. Here is an example error_log message for an SQL Injection attack:

```
[Tue Nov 16 16:02:38 2010] [error] [client ::1] ModSecurity: Access denied with
 code 403 (phase 2). Pattern match "\\bselect\\b.{0,40}\\buser\\b" at ARGS:foo.
[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 * from 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 "/vulnerable_app.php"]
[unique_id "TOLxbsCoC2oAABvWGW4AAAAA"]
```

This message format looks identical to the traditional rule logging format.

Pros and Cons of Traditional Detection Mode
Pros

- The benefit of this operating mode is that is much easier for a new user to understand.
- Better performance (lower latency/resources) as the first disruptive match will stop further processing.

Cons

Not optimal from a rules management perspective (handling false positives/exceptions)

- Editing a rule's complex Regular Expressions was difficult
- Typical method is to copy/paste the existing rule into a local custom rules file, edit the logic and then disable the existing CRS rule
- End result was that heavily customized rule sets were not updated when new CRS versions were released

Not optimal from a security perspective

- Not every site has the same risk tolerance
- Lower severity alerts are largely ignored
- Single low severity alerts may not be deemed critical enough to block, but multiple lower severity alerts in aggregate could be

Anomaly Scoring Detection Mode - Collaborative Rules Concept

In this advanced inspection mode, we are implementing the concept of ***Collaborative Detection*** and ***Delayed Blocking***. By this I mean that we have changed the rules logic by decoupling the inspection/detection from the blocking functionality. The individual rules can be run so that the detection remains, however instead of applying any disruptive action at that point, the rules will contribute to a transactional anomaly score collection. In addition, each rule will also store meta-data about each rule match (such as the Rule ID, Attack Category, Matched Location and Matched Data) within a unique TX variable.

Configuring Anomaly Scoring Detection Mode

If you want to run the CRS in Anomaly Scoring mode, you can do this easily by updating the SecDefaultAction line in the modsecurity_crs_10_config.conf file to use the pass action:

```
#
# -=[ Mode of Operation ]=-
#
# You can now choose how you want to run the modsecurity rules -
#
#       Anomaly Scoring vs. Traditional
#
# Each detection rule uses the "block" action which will inherit the SecDefaultAction
# specified below.  Your settings here will determine which mode of operation you use.
#
# Traditional mode is the current default setting and it uses "deny" (you can set any
# disruptive action you wish)
#
# If you want to run the rules in Anomaly Scoring mode (where blocking is delayed until the
# end of the request phase and rules contribute to an anomaly score) then set the
# SecDefaultAction to "pass"
#
# You can also decide how you want to handle logging actions.  You have three options -
#
#       - To log to both the Apache error_log and ModSecurity audit_log file use - log
#       - To log *only* to the ModSecurity audit_log file use - nolog,auditlog
#       - To log *only* to the Apache error_log file use - log,noauditlog
#
SecDefaultAction "phase:2,pass,log"
```

In this new mode, each matched rule will not block, but rather will increment anomaly scores using ModSecurity's setvar action. Here is an example of an SQL Injection CRS rule that is using setvar actions to increase both the overall anomaly score and the SQL Injection sub-category score:

```
 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,c
block,msg:'Blind SQL Injection Attack',id:'959514',tag:'WEB_ATTACK/SQL_INJECTION',tag:'WASCTC/WASC-19',tag:'OWASP_TOP_10/A1',tag
setvar:tx.sql_injection_score=+%{tx.critical_anomaly_score},setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},setvar:tx.%{ru
```

Anomaly Scoring Severity Levels

Each rule has a severity level specified. We have updated the rules to allow for the anomaly score collection incrementation to use macro expansion. Here is an example:

```
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}"
```

This allows the user to set their own anomaly score values from within the modsecurity_crs_10_config.conf file and these will be propagated out for use in the rules by using macro expansion.

```
#
# -=[ Anomaly Scoring Severity Levels ]=-
#
# These are the default scoring points for each severity level.  You may
# adjust these to you liking.  These settings will be used in macro expansion
# in the rules to increment the anomaly scores when rules match.
#
# These are the default Severity ratings (with anomaly scores) of the individual rules -
#
#    - 2: Critical - Anomaly Score of 5.
#         Is the highest severity level possible without correlation.  It is
#         normally generated by the web attack rules (40 level files).
#    - 3: Error - Anomaly Score of 4.
#         Is generated mostly from outbound leakage rules (50 level files).
#    - 4: Warning - Anomaly Score of 3.
#         Is generated by malicious client rules (35 level files).
#    - 5: Notice - Anomaly Score of 2.
#         Is generated by the Protocol policy and anomaly files.
#
```

```
SecAction "phase:1,t:none,nolog,pass, \
setvar:tx.critical_anomaly_score=5, \
setvar:tx.error_anomaly_score=4, \
setvar:tx.warning_anomaly_score=3, \
setvar:tx.notice_anomaly_score=2"
```

This configuration would mean that every CRS rule that has a Severity rating of "Critical" would increase the transactional anomaly score by 5 points per rule match. When we have a rule match, you can see how the anomaly scoring works from within the modsec_debug.log file:

```
Executing operator "rx" with param "\\bselect\\b.{0,40}\\buser\\b" against ARGS:foo.
Target value: "\xe2\x80\x98 union select * from user &#"
Added regex subexpression to TX.0: select * from user
Operator completed in 14 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:foo
Resolved macro %{tx.0} to: select * from user
Set variable "tx.959514-WEB_ATTACK/SQL_INJECTION-ARGS:foo" to "select * from user".
Resolved macro %{TX.0} to: select * from user
Warning. Pattern match "\bselect\b.{0,40}\buser\b" at ARGS:foo. [file "/usr/local/apache/conf/modsec_current/base_rules/modsecur
```

Anomaly Scoring Threshold Levels (Blocking)

Now that we have the capability to do anomaly scoring, the next step is to set our thresholds. This is the score value at which, if the current transactional score is above, it will be denied. We have two different anomaly scoring thresholds to set - one for the inbound request (which is evaluated at the end of phase:2 in the modsecurity_crs_49_inbound_blocking.conf file) and one for outbound information leakages (which is evaluated at the end of phase:4 in the modsecurity_crs_50_outbound_blocking.conf file):

```
#
# -=[ Anomaly Scoring Threshold Levels ]=-
#
# These variables are used in macro expansion in the 49 inbound blocking and 59
# outbound blocking files.
#
# **MUST HAVE** ModSecurity v2.5.12 or higher to use macro expansion in numeric
# operators.  If you have an earlier version, edit the 49/59 files directly to
# set the appropriate anomaly score levels.
#
# You should set the score to the proper threshold you would prefer. If set to "5"
# it will work similarly to previous Mod CRS rules and will create an event in the error_log
# file if there are any rules that match.  If you would like to lessen the number of events
# generated in the error_log file, you should increase the anomaly score threshold to
# something like "20".  This would only generate an event in the error_log file if
# there are multiple lower severity rule matches or if any 1 higher severity item matches.
#
SecAction "phase:1,t:none,nolog,pass,setvar:tx.inbound_anomaly_score_level=5"
SecAction "phase:1,t:none,nolog,pass,setvar:tx.outbound_anomaly_score_level=4"
```

With these current default settings, anomaly scoring mode will act similarly to traditional mode from a blocking perspective. Since all critical level rules increase the anomaly score by 5 points, this means that even 1 critical level rule match will cause a block. If you want to adjust the anomaly score so that you have a lower chance of blocking non-malicious clients (false positives) you could raise the tx.inbound_anomaly_score_level settings to something higher like 10 or 15. This would mean that two or more critical severity rules have matched before you decide to block. Another advantage of this approach is that you could aggregate multiple lower severity rule matches and then decide to block. So, one lower severity rule match (such as missing a Request Header such as Accept) would not result in a block but if multiple anomalies are triggered then the request would be blocked.

Enable/Disable Blocking
You are probably familiar with the SecRuleEngine directive which allows you to control blocking mode (On) vs. Detection mode (DetectionOnly). With the new Anomaly Scoring detection mode, if you want to allow blocking, you should set the SecRueEngine to On and then set the following TX variable in the modsecurity_crs_10_config.conf file:

```
#
# -=[ Anomaly Scoring Block Mode ]=-
#
# This is a collaborative detection mode where each rule will increment an overall
# anomaly score the transaction. The scores are then evaluated in the following files:
#
# Inbound anomaly score - checked in the modsecurity_crs_49_inbound_blocking.conf file
# Outbound anomaly score - checked in the modsecurity_crs_59_outbound_blocking.conf file
#
# If you do not want to use anomaly scoring mode, then comment out this line.
#
SecAction "phase:1,t:none,nolog,pass,setvar:tx.anomaly_score_blocking=on"
```

This is the rule within the modsecurity_crs_49_inbound_blocking.conf file that evaluates the anomaly scores at the end of the request phase and will block the request:

```
# Alert and Block based on Anomaly Scores
#
SecRule TX:ANOMALY_SCORE "@gt 0" \
    "chain,phase:2,t:none,deny,log,msg:'Inbound Anomaly Score Exceeded (Total Score: %{TX.ANOMALY_SCORE}, SQLi=%{TX.SQL_INJECTIC
        SecRule TX:ANOMALY_SCORE "@ge %{tx.inbound_anomaly_score_level}" chain
                SecRule TX:ANOMALY_SCORE_BLOCKING "@streq on" chain
                        SecRule TX:/^\d/ "(.*)"

# Alert and Block on a specific attack category such as SQL Injection
#
# SecRule TX:SQL_INJECTION_SCORE "@gt 0" \
#    "phase:2,t:none,log,block,msg:'SQL Injection Detected (score %{TX.SQL_INJECTION_SCORE}): %{tx.msg}'"
```

Notice that there is also another rule that is comment out by default. This example rule shows how you could alternatively choose to inspect/block based on a sub-category anomaly score (in the this example for SQL Injection). Let's take a look to see how this looks from within the modsec_debug.log file:

```
Recipe: Invoking rule 101a68700; [file "/usr/local/apache/conf/modsec_current/base_rules/modsecurity_crs_49_inbound_blocking.cor
Rule 101a68700: SecRule "TX:ANOMALY_SCORE" "@gt 0" "phase:2,log,chain,t:none,deny,msg:'Inbound Anomaly Score Exceeded (Total Scc
Transformation completed in 1 usec.
Executing operator "gt" with param "0" against TX:anomaly_score.
Target value: "10"
Operator completed in 3 usec.
Setting variable: tx.inbound_tx_msg=%{tx.msg}
Resolved macro %{tx.msg} to: SQL Injection Attack
Set variable "tx.inbound_tx_msg" to "SQL Injection Attack".
Setting variable: tx.inbound_anomaly_score=%{tx.anomaly_score}
Resolved macro %{tx.anomaly_score} to: 10
Set variable "tx.inbound_anomaly_score" to "10".
Rule returned 1.
Match -> mode NEXT_RULE.
Recipe: Invoking rule 101a761e0; [file "/usr/local/apache/conf/modsec_current/base_rules/modsecurity_crs_49_inbound_blocking.cor
Rule 101a761e0: SecRule "TX:ANOMALY_SCORE" "@ge %{tx.inbound_anomaly_score_level}" "chain"
Transformation completed in 0 usec.
Executing operator "ge" with param "%{tx.inbound_anomaly_score_level}" against TX:anomaly_score.
Target value: "10"
Resolved macro %{tx.inbound_anomaly_score_level} to: 5
Operator completed in 10 usec.
Rule returned 1.Match -> mode NEXT_RULE.
Recipe: Invoking rule 101a76880; [file "/usr/local/apache/conf/modsec_current/base_rules/modsecurity_crs_49_inbound_blocking.cor
Rule 101a76880: SecRule "TX:ANOMALY_SCORE_BLOCKING" "@streq on" "chain"
Transformation completed in 1 usec.
Executing operator "streq" with param "on" against TX:anomaly_score_blocking.
Target value: "on"
Operator completed in 1 usec.
Rule returned 1.Match -> mode NEXT_RULE.
Recipe: Invoking rule 101a76e28; [file "/usr/local/apache/conf/modsec_current/base_rules/modsecurity_crs_49_inbound_blocking.cor
Rule 101a76e28: SecRule "TX:/^\\d/" "@rx (.*)"
Expanded "TX:/^\d/" to "TX:0|TX:959514-WEB_ATTACK/SQL_INJECTION-ARGS:foo|TX:959047-WEB_ATTACK/SQL_INJECTION-ARGS:foo".
Transformation completed in 0 usec.
Executing operator "rx" with param "(.*)" against TX:0.
Target value: "union select"
Ignoring regex captures since "capture" action is not enabled.
Operator completed in 8 usec.
Rule returned 1.
Match, intercepted -> returning.
Resolved macro %{TX.ANOMALY_SCORE} to: 10
Resolved macro %{TX.SQL_INJECTION_SCORE} to: 10
Resolved macro %{tx.msg} to: SQL Injection Attack
Resolved macro %{matched_var} to: union select
Access denied with code 403 (phase 2). Pattern match "(.*)" at TX:0. [file "/usr/local/apache/conf/modsec_current/base_rules/moc
[msg "Inbound Anomaly Score Exceeded (Total Score: 10, SQLi=10, XSS=): Last Matched Message: SQL Injection Attack"] [data "Last
```

Alert Management - Correlated Events

The CRS events in the Apache error_log file can become very chatty. This is due to running the CRS in traditional detection mode where each rule is triggering its own error_log entry. What would be more useful for the security analyst would be for only 1 correlated event to be generated and logged to the Apache error_log file that would give the user a higher level determination of the transaction severity.

To achieve this capability, the CRS can be run in Anomaly Scoring mode where each individual rule will generate an audit log event Message entry but they will not log to the error_log on their own. These rules are considered basic or reference events (that have contributed to the overall anomaly score) and may be reviewed in the audit log if the user wants to see what individual events contributed to the overall anomaly score and event designation. To configure this capability, simply edit the SecDefaultAction line in the modsecurity_crs_10_config.conf file like this:

```
#
# -=[ Mode of Operation ]=-
#
# You can now choose how you want to run the modsecurity rules -
#
#       Anomaly Scoring vs. Traditional
#
# Each detection rule uses the "block" action which will inherit the SecDefaultAction
# specified below.  Your settings here will determine which mode of operation you use.
#
# Traditional mode is the current default setting and it uses "deny" (you can set any
# disruptive action you wish)
#
```

```
# If you want to run the rules in Anomaly Scoring mode (where blocking is delayed until the
# end of the request phase and rules contribute to an anomaly score) then set the
# SecDefaultAction to "pass"
#
# You can also decide how you want to handle logging actions.  You have three options -
#
#       - To log to both the Apache error_log and ModSecurity audit_log file use - log
#       - To log *only* to the ModSecurity audit_log file use - nolog,auditlog
#       - To log *only* to the Apache error_log file use - log,noauditlog
#
SecDefaultAction "phase:2,pass,nolog,auditlog"
```

With this setting, rule matches will log the standard Message data to the modsec_audit.log file. You will still get 1 correlated event logged to the normal Apache error_log file from the rules within the modsecurity_crs_49_inbound_blocking.conf file. The resulting Apache error_log entry would look like this:

```
[Wed Nov 17 11:46:56 2010] [error] [client ::1] ModSecurity: Access denied with code 403 (phase 2). Pattern match "(.*)" at TX:0
[msg "Inbound Anomaly Score Exceeded (Total Score: 10, SQLi=10, XSS=): Last Matched Message: SQL Injection Attack"] [data "Last
```

This entry tells us that there was an SQL Injection attack identified on the inbound request. We see that Total anomaly score is 10 and that the sub-category score of SQLi is 10. This tells us that there were 2 different SQL Injection rules that triggered. If you want to see the details of all of the reference events (individual rules that contributed to this correlated event), you can review the modsec_audit.log data for this transaction. Section H of the audit log shows rule matches:

```
--0a4c3b0e-A--
[17/Nov/2010:11:46:56 --0500] TOQHAMCoAWcAAB7KH10AAAAB ::1 49415 ::1 80
--0a4c3b0e-B--
GET /vulnerable_app.php?foo=%E2%80%98+UNION+SELECT+*+FROM+user+%26%23 HTTP/1.1
Host: localhost
Connection: keep-alive
Cache-Control: max-age=0
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 Saf
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

--0a4c3b0e-F--
HTTP/1.1 403 Forbidden
Content-Length: 220
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1

--0a4c3b0e-H--
Message: Pattern match "\bselect\b.{0,40}\buser\b" at ARGS:foo. [file "/usr/local/apache/conf/modsec_current/base_rules/modsecur
Message: Pattern match "\bunion\b.{1,100}?\bselect\b" at ARGS:foo. [file "/usr/local/apache/conf/modsec_current/base_rules/modse
Message: Access denied with code 403 (phase 2). Pattern match "(.*)" at TX:0. [file "/usr/local/apache/conf/modsec_current/base_
Message: Warning. Operator GE matched 5 at TX:inbound_anomaly_score. [file "/usr/local/apache/conf/modsec_current/base_rules/moc
Action: Intercepted (phase 2)
Stopwatch: 1290012416382280 122228 (8369 120370 -)
Producer: ModSecurity for Apache/2.5.13dev2 (http://www.modsecurity.org/); core ruleset/2.0.9.
Server: Apache/2.2.12 (Unix) mod_ssl/2.2.12 OpenSSL/0.9.8l DAV/2
--0a4c3b0e-Z--
```

By reviewing the audit log entry for this event, you can see all of the details of the two different SQL Injection rules that triggered on this request and contributed to the anomaly score.

Pros and Cons of Anomaly Scoring Detection Mode

Pros

- An increased confidence in blocking - since more detection rules contribute to the anomaly score, the higher the score, the more confidence you can have in blocking malicious transactions.
- Allows users to set a threshold that is appropriate for them - different sites may have different thresholds for blocking.
- Allows several low severity events to trigger alerts while individual ones are suppressed.
- One correlated event helps alert management.
- ***Exceptions may be handled by either increasing the overall anomaly score threshold, or by adding rules to a local custom exceptions file where TX data of previous rule matches may be inspected and anomaly scores re-adjusted based on the false positive criteria***.

Cons

- More complex for the average user.
- Log monitoring scripts may need to be updated for proper analysis

Posted by Nicholas J. Percoco on 17 November 2010 at 14:41 in ModSecurity | Permalink ShareThis

TrackBack

TrackBack URL for this entry:
http://www.typepad.com/services/trackback/6a0133f264aa62970b0134896e2c00970c
Listed below are links to weblogs that reference Advanced Topic of the Week: Traditional vs. Anomaly Scoring Detection Modes:

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