1 Preparing the Battle Space

The art of war teaches us to rely not on the likelihood of the enemy's not coming, but on our own readiness to receive him; not on the chance of his not attacking, but rather on the fact that we have made our position unassailable.

—Sun Tzu in *The Art of War*

"Is our web site secure?" If your company's chief executive officer asked you this question, what would you say? If you respond in the affirmative, the CEO might say, "Prove it." How do you provide tangible proof that your web applications are adequately protected? This section lists some sample responses and highlights the deficiencies of each. Here's the first one:

> *Our web applications are secure because we are compliant with the Payment Card Industry Data Security Standard (PCI DSS).*

PCI DSS, like most other regulations, is a *minimum standard of due care*. This means that achieving compliance does not make your site unhackable. PCI DSS is really all about risk transference (from the credit card companies to the merchants) rather than risk mitigation. If organizations do not truly embrace the concept of reducing risk by securing their environments above and beyond what PCI DSS specifies, the compliance process becomes nothing more than a checkbox paperwork exercise. Although PCI has some admirable aspects, keep this mantra in mind:

> *It is much easier to pass a PCI audit if you are secure than to be secure because you pass a PCI audit.*

In a more general sense, regulations tend to suffer from the *control-compliant* philosophy. They are input-centric and do not actually analyze or monitor their effectiveness in operations. Richard

Bejtlich,[1] a respected security thought leader, brilliantly presented this interesting analogy on this topic:

> *Imagine a football (American-style) team that wants to measure their success during a particular season. Team management decides to measure the height and weight of each player. They time how fast the player runs the 40 yard dash. They note the college from which each player graduated. They collect many other statistics as well, then spend time debating which ones best indicate how successful the football team is. Should the center weigh over 300 pounds? Should the wide receivers have a shoe size of 11 or greater? Should players from the northwest be on the starting lineup? All of this seems perfectly rational to this team. An outsider looks at the situation and says: "Check the scoreboard! You're down 42–7 and you have a 1–6 record. You guys are losers!"*

This is the essence of input-centric versus output-aware security. Regardless of all the preparations, it is on the live production network where all your web security preparations will either pay off or crash and burn. Because development and staging areas rarely adequately mimic production environments, you do not truly know how your web application security will fare until it is accessible by untrusted clients.

> *Our web applications are secure because we have deployed commercial web security product(s).*

This response is an unfortunate result of transitive belief in security. Just because a security vendor's web site or product collateral says that the product will make your web application more secure does not in fact make it so. Security products, just like the applications they are protecting, have flaws if used incorrectly. There are also potential issues with mistakes in configuration and deployment, which may allow attackers to manipulate or evade detection.

> *Our web applications are secure because we use SSL.*

Many e-commerce web sites prominently display an image seal. This indicates that the web site is secure because it uses a Secure Socket Layer (SSL) certificate purchased from a reputable certificate authority (CA). Use of an SSL signed certificate helps prevent the following attacks:

- **Network sniffing.** Without SSL, your data is sent across the network using an unencrypted channel. This means that anyone along the path can potentially sniff the traffic off the wire in clear text.
- **Web site spoofing.** Without a valid SSL site certificate, it is more difficult for attackers to attempt to use phishing sites that mimic the legitimate site.

The use of SSL does help mitigate these two issues, but it has one glaring weakness: *The use of SSL does absolutely nothing to prevent a malicious user from directly attacking the web application itself.* As a matter of fact, many attackers prefer to target SSL-enabled web applications because using this encrypted channel may hide their activities from other network-monitoring devices.

Our web applications are secure because we have alerts demonstrating that we blocked web attacks.

Evidence of blocked attack attempts is good but is not enough. When management asks if the web site is secure, it is really asking what the score of the game is. The CEO wants to know whether you are winning or losing the game of defending your web applications from compromise. In this sense, your response doesn't answer the question. Again referencing Richard Bejtlich's American football analogy, this is like someone asking you who won the Super Bowl, and you respond by citing game statistics such as number of plays, time of possession, and yards gained without telling him or her the final score! Not really answering the question is it? Although providing evidence of blocked attacks is a useful metric, management really wants to know if any successful attacks occurred.

With this concept as a backdrop, here are the web security metrics that I feel are most important for the production network and gauging how the web application's security mechanisms are performing:

- **Web transactions per day** should be represented as a number (#). It establishes a baseline of web traffic and provides some perspective for the other metrics.
- **Attacks detected (true positive)** should be represented as both a number (#) and a percentage (%) of the total web transactions per day. This data is a general indicator of both malicious web traffic and security detection accuracy.
- **Missed attacks (false negative)** should be represented as both a number (#)and a percentage (%) of the total web transactions per day. This data is a general indicator of the effectiveness of security detection accuracy. This is the key metric that is missing when you attempt to provide the final score of the game.
- **Blocked traffic (false positive)** should be represented as both a number (#) and a percentage (%) of the total web transactions per day. This data is also a general indicator of the effectiveness of security detection accuracy. This is very important data for many organizations because blocking legitimate customer traffic may mean missed revenue. Organizations should have a method of accurately tracking false positive alerts that took disruptive actions on web transactions.
- **Attack detection failure rate** should be represented as a percentage (%). It is derived by adding false negatives and false positives and then dividing by true

positives. *This percentage gives the overall effectiveness of your web application security detection accuracy.*

The attack detection failure rate provides data to better figure out the score of the game. Unfortunately, most organizations do not gather enough information to conduct this type of security metric analysis.

Our web applications are secure because we have not identified any abnormal behavior.

From a compromise perspective, identifying abnormal application behavior seems appropriate. The main deficiency with this response has to do with the data used to identify anomalies. Most organizations have failed to properly instrument their web applications to produce sufficient logging detail. Most web sites default to using the web server's logging mechanisms, such as the Common Log Format (CLF). Here are two sample CLF log entries taken from the Apache web server:

```
109.70.36.102 - - [15/Feb/2012:09:08:16 -0500] "POST /wordpress//xmlrpc.php
 HTTP/1.1"
500 163 "-" "Wordpress Hash Grabber v2.0libwww-perl/6.02"
109.70.36.102 - - [15/Feb/2012:09:08:17 -0500] "POST /wordpress//xmlrpc.php
 HTTP/1.1"
200 613 "-" "Wordpress Hash Grabber v2.0libwww-perl/6.02"
```

Looking at this data, we can see a few indications of potential suspicious or abnormal behavior. The first is that the User-Agent field data shows a value for a known WordPress exploit program, WordPress Hash Grabber. The second indication is the returned HTTP status code tokens. The first entry results in a 500 Internal Server Error status code, and the second entry results in a 200 OK status code. What data in the first entry caused the web application to generate an error condition? We don't know what parameter data was sent to the application because `POST` requests pass data in the request body rather than in a `QUERY_STRING` value that is logged by web servers in the CLF log. What data was returned within the response bodies of these transactions? These are important questions to answer, but CLF logs include only a small subset of the full transactional data. They do not, for instance, include other request headers such as cookies, POST request bodies, or any logging of outbound data. Failure to properly log outbound HTTP response data prevents organizations from answering this critical incident response question: "What data did the attackers steal?" The lack of robust HTTP audit logs is one of the main reasons why organizations cannot conduct proper incident response for web-related incidents.

Our web applications are secure because we have not identified any abnormal behavior, and we collect and analyze full HTTP audit logs for signs of malicious behavior.

A key mistake that many organizations make is to use only alert-centric events as indicators of potential incidents. If you log only details about known malicious behaviors, how will you know if your defenses are ever circumvented? New or stealthy attack methods emerge constantly. Thus, it is insufficient to analyze alerts for issues you already know about. You must have full HTTP transactional audit logging at your disposal so that you may analyze them for other signs of malicious activity.

During incident response, management often asks, "What else did this person do?" To accurately answer this question, you must have audit logs of the user's entire web session, not just a single transaction that was flagged as suspicious.

> *Our web applications are secure because we have not identified any abnormal behavior, and we collect and analyze full HTTP audit logs for signs of malicious behavior. We also regularly test our applications for the existence of vulnerabilities.*

Identifying and blocking web application attack attempts is important, but correlating the target of these attacks with the existence of known vulnerabilities within your applications is paramount. Suppose you are an operational security analyst for your organization who manages events that are centralized within a Security Information Event Management (SIEM) system. Although a spike in activity for attacks targeting a vulnerability within a Microsoft Internet Information Services (IIS) web server indicates malicious behavior, the severity of these alerts may be substantially less if your organization does not use the IIS platform. On the other hand, if you see attack alerts for a known vulnerability within the osCommerce application, and you are running that application on the system that is the target of the alert, the threat level should be increased, because a successful compromise is now a possibility. Knowing which applications are deployed in your environment and if they have specific vulnerabilities is critical for proper security event prioritization. Even if you have conducted full application assessments to identify vulnerabilities, this response is still incomplete, and this final response highlights why:

> *Our web applications are secure because we have not identified any abnormal behavior, and we collect and analyze full HTTP audit logs for signs of malicious behavior. We also regularly test our applications for the existence of vulnerabilities and our detection and incident response capabilities.*

With this final response, you see why the preceding answer is incomplete. Even if you know where your web application vulnerabilities are, you still must actively test your operational security defenses with live simulations of attacks to ensure their effectiveness.

Does operational security staff identify the attacks? Are proper incident response countermeasures implemented? How long does it take to implement them? Are the countermeasures effective? Unless you can answer these questions, you will never truly know if your defensive mechanisms are working.

[1] http://taosecurity.blogspot.com/

1 Application Fortification

Whoever is first in the field and awaits the coming of the enemy will be fresh for the fight; whoever is second in the field and has to hasten to battle will arrive exhausted.

—Sun Tzu in *The Art of War*

The recipes in this section walk you through the process of preparing your web application for the production network battlefront.

The first step is application fortification, in which you analyze the current web application that you must protect and enhance its defensive capabilities.

RECIPE 1-1: REAL-TIME APPLICATION PROFILING

This recipe shows you how to use ModSecurity's Lua API to analyze HTTP transactions to develop a learned profile of expected request characteristics.

Ingredients

- ModSecurity Reference Manual[1]
  - Lua API support
  - `SecRuleScript` directive
  - `initcol` action
  - `RESOURCE` persistent storage
- OWASP ModSecurity Core Rule Set[2]
  - modsecurity_crs_40_appsensor_detection_point_2.0_setup.conf
  - modsecurity_crs_40_appsensor_detection_point_2.1_request_exception.conf
  - modsecurity_crs_40_appsensor_detection_point_3.0_end.conf
  - appsensor_request_exception_profile.lua

Learning about Expected Request Attributes

The concepts in this section demonstrate how to dynamically create a positive security model or whitelisting input validation envelope around your application. After it is created, this external security layer will enforce rules for the critical elements of an application and allow you to identify abnormal requests that violate this policy. This recipe shows how to profile real web user transactions to identify the following request attributes for each application resource:

- Request method(s)
- Number of parameters (minimum/maximum range)
- Parameter names
- Parameter lengths (minimum/maximum range)
- Parameter types
  - Flag (such as /path/to/foo.php?param)
  - Digits (such as /path/to/foo.php?param=1234)
  - Alpha (such as /path/to/foo.php?param=abcd)
  - Alphanumeric (such as /path/to/foo.php?param=abcd1234)
  - E-mail (such as /path/to/foo.php?param=foo@bar.com)
  - Path (such as /path/to/foo.php?param=/dir/somefile.txt)
  - URL (such as /path/to/foo.php?param=http://somehost/dir/file.txt)
  - SafeText (such as /path/to/foo.php?param=some_data-12)

NOTE

Why is an external input validation layer needed? One key paradigm with web application development is at the core of the majority of vulnerabilities we face: *Web developers do not have control of the web client software*. Think about this for a moment, because it is the lynchpin theory on which most of our problems rest. Web developers frequently assume that data that is sent to the client is immune to manipulation. They believe this is true either because of web browser user interface restrictions (such as data being stored hidden in fields or drop-down lists) or because of the implementation of security controls using browser-executed code such as JavaScript. With this false belief in place, web developers incorrectly assume that certain data within web requests has not been modified, and they simply accept the input for execution within server-side processing. In reality, it is easy for attackers to bypass web browser security controls either by using custom browser plug-ins or by using a client-side intercepting proxy. With these tools in place, attackers may easily bypass any client-side security code and manipulate any HTTP data that is being sent to or from their browsers.

Creating Persistent Storage

With ModSecurity, we can leverage per-resource persistent storage so that we can correlate data across multiple requests and clients. We do this by initializing the RESOURCE persistent storage mechanism early in the request phase (phase:1 in ModSecurity's transactional hooks):

```
#
# --[ Step 1: Initiate the Resource Collection ]--
#
# We are using the REQUEST_FILENAME as the key and then set 2
# variables -
#
# [resource.pattern_threshold]
# Set the resource.pattern_threshold as the minimum number of times
# that a match should occur in order to include it into the profile
#
# [resource.confidence_counter_threshold]
# Set the resource.confidence_counter_threshold as the minimum number
# of "clean" transactions to profile/inspect before enforcement of
# the profile begins.
#
SecAction \
"phase:1,id:'981082',t:none,nolog,pass,\
initcol:resource=%{request_headers.host}_%{request_filename},\
setvar:resource.min_pattern_threshold=9, \
setvar:resource.min_traffic_threshold=100"
```

The `initcol:resource` action uses the macro expanded `REQUEST_HEADERS:Host` and `REQUEST_FILENAME` variables as the collection key to avoid any potential collisions with similarly named resources. The two `setvar` actions are used to determine the number of transactions we want to profile and how many times our individual checks must match before we add them to the enforcement list.

Post-Process Profiling

We want to minimize the potential latency impact of this profiling analysis so it is conducted within the post-processing phase after the HTTP response has already gone out to the client (phase:5 in ModSecurity). Before we decide whether to profile the transaction, we want to do a few security checks to ensure that we are looking at only "clean" transactions that are free from malicious input. This is important because we don't want to include attack data within our learned profile.

```
#
# --[ Begin Profiling Phase ]--
#
```

```
SecMarker BEGIN_RE_PROFILE_ANALYSIS

SecRule RESPONSE_STATUS "^404$" \
"phase:5,id:'981099',t:none,nolog,pass,setvar:!resource.KEY,\
skipAfter:END_RE_PROFILE_ANALYSIS"
SecRule RESPONSE_STATUS "^(5|4)" \
 "phase:5,id:'981100',t:none,nolog,pass, \
skipAfter:END_RE_PROFILE_ANALYSIS"
SecRule TX:ANOMALY_SCORE "!@eq 0" \
 "phase:5,id:'981101',t:none,nolog,pass, \
skipAfter:END_RE_PROFILE_ANALYSIS"
SecRule &RESOURCE:ENFORCE_RE_PROFILE "@eq 1" \
"phase:2,id:'981102',t:none,nolog,pass, \
skipAfter:END_RE_PROFILE_ANALYSIS"
SecRuleScript crs/lua/appsensor_request_exception_profile.lua \
"phase:5,nolog,pass"

SecMarker END_RE_PROFILE_ANALYSIS
```

There are four different transactional scenarios in which we don't want to profile the data:

- If the HTTP response code is 404, the resource doesn't exist. In this case, not only do we skip the profiling, but we also remove the resource key, so we delete the persistent storage. This is achieved by using the `setvar:!resource.KEY` action.
- If the HTTP response code is either level 4*xx* or level 5*xx*, the application says something is wrong with the transaction, so we won't profile it in this case either.
- The OWASP ModSecurity Core Rule Set (CRS), which we will discuss in Recipe 1-3, can use anomaly scoring. We can check this transactional anomaly score. If it is anything other than 0, we should skip profiling.
- Finally, we have already seen enough traffic for our profiling model and are currently in *enforcement mode*, so we skip profiling.

If all these prequalifier checks pass, we then move to the actual profiling of the request attributes by using the appsensor_request_exception_profile.lua script, which is called by the `SecRuleScript` directive.

Sample Testing

To test this profiling concept, let's look at a sample resource from the WordPress application in which a client can submit a comment to the blog. Figure 1-1 shows a typical form in which a user can leave a reply.

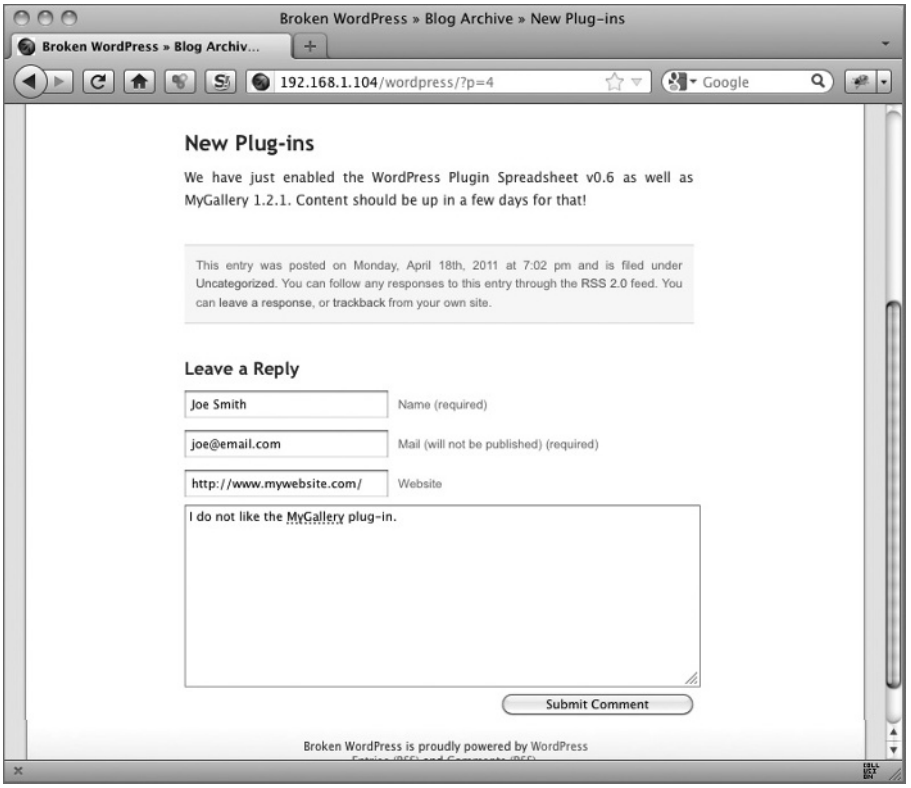

Figure 1-1: WordPress Leave a Reply form

When the client clicks the Submit Comment button, this is how the HTTP request looks when the web application receives it:

```
POST /wordpress/wp-comments-post.php HTTP/1.1
Host: 192.168.1.104
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:10.0.1)
Gecko/20100101 Firefox/10.0.1
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
DNT: 1
Connection: keep-alive
Referer: http://192.168.1.104/wordpress/?p=4
Content-Type: application/x-www-form-urlencoded
Content-Length: 161

author=Joe+Smith&email=joe%40email.com&url=http%3A%2F%2Fwww.mywebsite
.com%2F&comment=I+do+not+like+the+MyGallery+plug-in.
&submit=Submit+Comment&comment_post_ID=4
```

We can see that the `REQUEST_METHOD` is `POST` and that six parameters are sent within the `REQUEST_BODY` payload:

- `author` is the name of the person submitting the comment. This value should be different for each user.
- `email` is the e-mail address of the person submitting the comment. This value should be different for each user.
- `url` is the web site associated with the user submitting the comment. This value should be different for each user.
- `comment` is a block of text holding the actual comment submitted by the user. This value should be different for each user.
- `submit` is a static payload and should be the same for all users.
- `comment_post_ID` is a numeric field holding a value that is unique for each comment post.

I sent the following two requests to the WordPress application using the curl HTTP client tool to simulate user traffic with different sizes of payloads:

```
$ for f in {1..50} ; do curl -H "User-Agent: Mozilla/5.0 (Macintosh;
Intel Mac OS X 10.6; rv:10.0.1) Gecko/20100101 Firefox/10.0.1" -d
"author=Joe+Smith$f&email=joe%40email$f.com&url=http%3A%2F%2Fwww.
mywebsite$f.com%2F&comment=I+do+not+like+the+MyGallery+plug-
in.$f&submit=Submit+Comment&comment_post_ID=$f" "http://localhost/
wordpress/wp-comments-post.php" ; done

$ for f in {100..151} ; do curl -H "User-Agent: Mozilla/5.0 (Macintosh
; Intel Mac OS X 10.6; rv:10.0.1) Gecko/20100101 Firefox/10.0.1" -d
 "author=Jane+Johnson$f&email=jane.johnson%40cool-email$f.com
&url=http%3A%2F%2Fwww.someotherwebsite$f.com%2F&comment=I+do+LOVE+the
+MyGallery+plug-in.+It+shows+cool+pix.$f&submit=Submit+Comment&
comment_post_ID=$f" http://localhost/wordpress/wp-comments-post.php
 ; done
```

These requests cause the Lua script to profile the request characteristics and save this data in the resource persistent collection file. When the number of transactions profiled reaches the confidence counter threshold (in this case, 100 transactions), the script adds the *enforcement* variable. This causes the Lua script to stop profiling traffic for this particular resource and activates the various enforcement checks for any subsequent inbound requests. When the profiling phase is complete, we can view the saved data in resource persistent storage by using a Java tool called jwall-tools written by Christian Bockermann.

Here is a sample command that is piped to the `sed` command for easier output for this book:

```
$ sudo java -jar jwall-tools-0.5.3-5.jar collections -a -s -l /tmp |
sed -e 's/^.*]\.//g'
Reading collections from /tmp
```

```
Collection resource, last read @ Thu Feb 16 20:04:54 EST 2012
   Created at Thu Feb 16 16:59:54 EST 2012
ARGS:author_length_10_counter = 9
ARGS:author_length_11_counter = 41
ARGS:author_length_15_counter = 2
ARGS:comment_length_37_counter = 9
ARGS:comment_length_38_counter = 41
ARGS:comment_length_54_counter = 2
ARGS:comment_post_ID_length_1_counter = 9
ARGS:comment_post_ID_length_2_counter = 41
ARGS:comment_post_ID_length_3_counter = 2
ARGS:email_length_14_counter = 9
ARGS:email_length_15_counter = 41
ARGS:email_length_30_counter = 2
ARGS:submit_length_14_counter = 2
ARGS:url_length_26_counter = 9
ARGS:url_length_27_counter = 41
ARGS:url_length_35_counter = 2
MaxNumOfArgs = 6
MinNumOfArgs = 6
NumOfArgs_counter_6 = 2
TIMEOUT = 3600
enforce_ARGS:author_length_max = 15
enforce_ARGS:author_length_min = 10
enforce_ARGS:comment_length_max = 54
enforce_ARGS:comment_length_min = 37
enforce_ARGS:comment_post_ID_length_max = 3
enforce_ARGS:comment_post_ID_length_min = 1
enforce_ARGS:email_length_max = 30
enforce_ARGS:email_length_min = 14
enforce_ARGS:submit_length_max = 14
enforce_ARGS:submit_length_min = 14
enforce_ARGS:url_length_max = 35
enforce_ARGS:url_length_min = 26
enforce_args_names = author, email, url, comment, submit,
comment_post_ID
enforce_charclass_digits = ARGS:comment_post_ID
enforce_charclass_email = ARGS:email
enforce_charclass_safetext = ARGS:author, ARGS:comment, ARGS:submit
enforce_charclass_url = ARGS:url
enforce_num_of_args = 6
enforce_re_profile = 1
enforce_request_methods = POST
min_pattern_threshold = 9
min_traffic_threshold = 100
request_method_counter_POST = 2
traffic_counter = 102
This collection expired 2h 2m 56.485s seconds ago.
```

As you can see, we can now validate a number of request attributes on future requests. Chapter 5 includes examples of using this profile to identify anomalies and attacks.

CAUTION

This recipe is a great first step in providing input validation through profiling of real traffic, but it is not perfect. It has three main limitations.

No Auto-Relearning

The main limitation of this specific implementation is that it offers no *auto-relearning*. As soon as the rules have moved from the profiling phase into the enforcement phase, the implementation stays in that mode. This means that if you have a legitimate code push that updates functionality to an existing resource, you will probably have to remove the resource collection and then begin learning again.

Persistent Storage Size Limitations

Depending on the number of profile characteristics you need to profile per resource, you may run into SDBM persistent storage size limits. By default, approximately 800 bytes of usable storage is available in the ModSecurity persistent storage files. If you run into this issue, you see this error message:

```
[1] Failed to write to DBM file "/tmp/RESOURCE": Invalid argument
```

In this case, you need to update the default storage limit available to you in the Apache Portable Runtime (APR) package. If you need more than that, you should download the separate APR and APR-Util packages and then edit the `#define PAIRMAX` setting in the /dbm/sdbm/sdbm_private.h file:

```
/* if the block/page size is increased, it breaks perl apr_sdbm_t
* compatibility */
#define DBLKSIZ 16384
#define PBLKSIZ 8192
#define PAIRMAX 8008                    /* arbitrary on PBLKSIZ-N */
#else
#define DBLKSIZ 16384
#define PBLKSIZ 8192
#define PAIRMAX 10080                   /* arbitrary on PBLKSIZ-N */
#endif
#define SPLTMAX 10
```

You should then recompile both Apache and ModSecurity, referencing the updated APR/APR-Util packages.

Excluding Resources

If you want to exclude certain URLs from profiling, you can activate some commented-out rules. They do an `@pmFromFile` check against an external file. This allows you to add URLs to be excluded to this list file.

[1] http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual

[2] https://www.owasp.org/index.php Category:OWASP_ModSecurity_Core_Rule_Set_Project

RECIPE 1-2: PREVENTING DATA MANIPULATION WITH CRYPTOGRAPHIC HASH TOKENS

This recipe shows you how to use ModSecurity to implement additional hash tokens to outbound HTML data to prevent data manipulation attacks. When this book was written, the capabilities outlined in this recipe were available in ModSecurity version 2.7. Future versions may have different or extended functionality.

Ingredients

- ModSecurity Reference Manual[3]
  - Version 2.7 or higher
  - `SecDisableBackendCompression` directive
  - `SecContentInjection` directive
  - `SecStreamOutBodyInspection` directive
  - `SecEncryptionEngine` directive
  - `SecEncryptionKey` directive
  - `SecEncryptionParam` directive
  - `SecEncryptionMethodRx` directive

As mentioned earlier, web developers cannot rely on web browser security mechanisms to prevent data manipulation. With this being the case, how can we implement an external method of verifying that outbound data has not been manipulated when returned in a follow-up request? One technique is to parse the outbound HTML response body data and inject additional token data into select locations. The data we are injecting is called request parameter validation tokens. These are essentially cryptographic hashes of select HTML page elements. The hashes enable us to detect if the client attempts to tamper with the data. Here are some sample ModSecurity directives and rules that implement basic hashing protections:

```
SecDisableBackendCompression On
SecContentInjection On
SecStreamOutBodyInspection On
SecEncryptionEngine On
SecEncryptionKey rand keyOnly
SecEncryptionParam rv_token
```

```
SecEncryptionMethodrx "HashUrl" "[a-zA-Z0-9]"
SecRule REQUEST_URI "@validateEncryption [a-zA-Z0-9]" "phase:2,
id:1000,t:none,block,msg:'Request Validation Violation.',
ctl:encryptionEnforcement=On"
```

The first directive, `SecDisableBackendCompression`, is needed only in a reverse proxy setup. It is used if the web application is compressing response data in the gzip format. This is needed so that we can parse the response HTML data and modify it. ModSecurity's default configuration is to make copies of transactional data in memory and inspect them while buffering the real connection. The next two directives are used together, however, so that the original buffered response body can be modified and replaced with the new one. The next four `SecEncryption` directives configure the basic settings. In this configuration, ModSecurity uses a random encryption key as the hash salt value and hashes HTML `href` components that match the defined regular expression. The final `SecRule` is used to validate and enforce the hash tokens.

Let's look at a practical example. Figure 1-2 shows a section of HTML from a WordPress page that includes an `href` hyperlink. This link includes both a universal resource identifier (URI) and a query string value with a parameter named `p` with a numeric character value of 4.

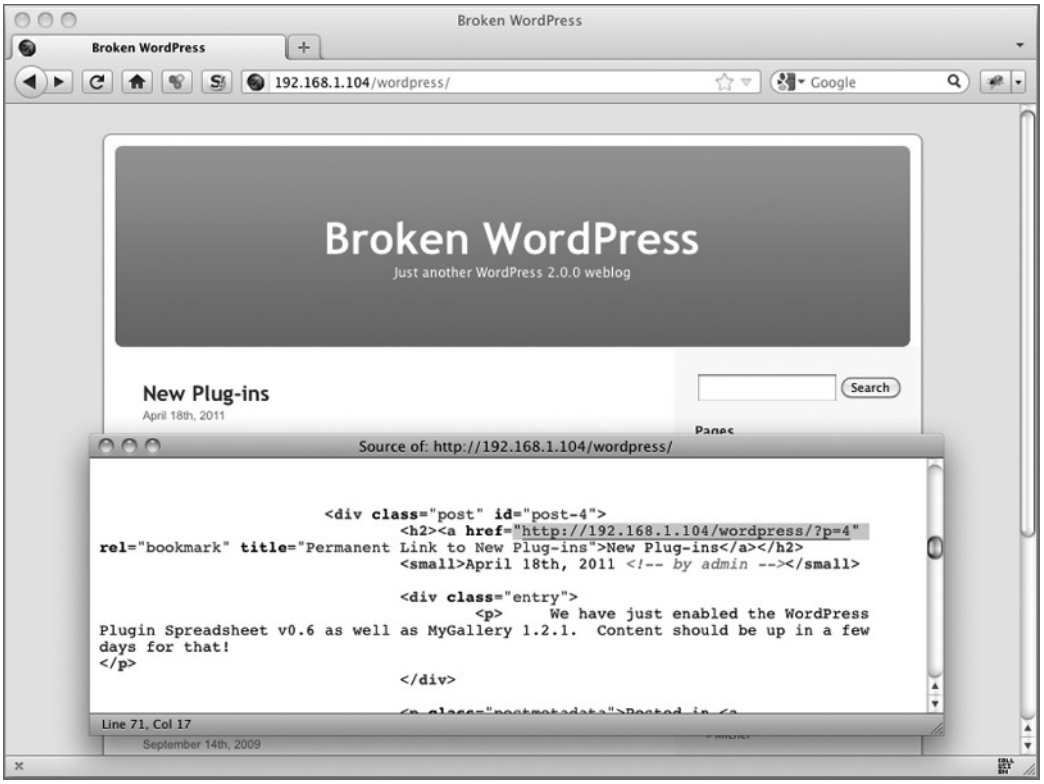

Figure 1-2: WordPress HTML source showing an href link with a parameter

After the encryption rules are put in place, ModSecurity parses the outbound HTML data and searches for elements to which the hash tokens can be added based on the protection configuration. Here is an example from the debug log file showing the hashing process internals:

```
Output filter: Completed receiving response body (buffered full-
 8202 bytes).
init_response_body_html_parser: Charset[UTF-8]
init_response_body_html_parser: Successfully html parser
generated.
Signing data [feed://http:/192.168.1.104/wordpress/?feed=comments
-rss2]
Signing data [feed://http:/192.168.1.104/wordpress/?feed=rss2]
Signing data [xfn/]
Signing data [check/referer]
Signing data [wordpress/wp-login.php]
Signing data [wordpress/wp-register.php]
Signing data [weblog/]
Signing data [journalized/]
Signing data [xeer/]
Signing data [wordpress/?cat=1]
Signing data [wordpress/?m=200909]
Signing data [wordpress/?m=201104]
Signing data [wordpress/?page_id=2]
Signing data [wordpress/?p=1#comments]
Signing data [wordpress/?cat=1]
Signing data [wordpress/?p=1]
Signing data [wordpress/?p=3#comments]
Signing data [wordpress/?cat=1]
Signing data [wordpress/?p=3]
Signing data [wordpress/?p=4#respond]
Signing data [wordpress/?cat=1]
Signing data [wordpress/?p=4]
Signing data [wordpress/]
Signing data [wordpress/xmlrpc.php?rsd]
Signing data [wordpress/xmlrpc.php]
Signing data [wordpress/?feed=rss2]
Signing data [wordpress/wp-content/themes/default/style.css]
encrypt_response_body_links: Processed [0] iframe src, [0]
encrypted.
encrypt_response_body_links: Processed [0] frame src, [0]
encrypted.
encrypt_response_body_links: Processed [0] form actions, [0]
encrypted.
encrypt_response_body_links: Processed [33] links, [27]
encrypted.
inject_encrypted_response_body: Detected encoding type [UTF-8].
inject_encrypted_response_body: Using content-type [UTF-8].
inject_encrypted_response_body: Copying XML tree from CONV to
stream buffer [8085] bytes.
```

```
inject_encrypted_response_body: Stream buffer [8750]. Done
Encryption completed in 2829 usec.
```

Upon completion, all `href` link data is updated to include a new `rv_token` that contains a hash of the full URI (including any query string parameter data), as shown in Figure 1-3.

With this protection in place, any modifications to either the URI or the parameter payload result in ModSecurity alerts (and blocking depending on your policy settings). The following sections describe the two sample attack scenarios you will face.

Figure 1-3: WordPress HTML source showing an updated href link with rv_token data

Hash Token Mismatches

If an attacker attempts to modify the parameter data, ModSecurity generates an alert. For example, if the attacker inserts a single quote character (which is a common way to test for SQL Injection attacks), the following alert is generated:

```
Rule 100909d20: SecRule "REQUEST_URI" "@validateEncryption
[a-zA-Z0-9]" "phase:2,log,id:1000,t:none,block,msg:'Request
Validation Violation.',ctl:encryptionEnforcement=On"
```

```
Transformation completed in 1 usec.
Executing operator "validateEncryption" with param "[a-zA-Z0-9]"
against REQUEST_URI.
Target value: "/wordpress/?p=4%27&rv_token=
aafb509403bbf7d78c3c8fe1735d49f01b90eb64"
Signing data [wordpress/?p=4%27]Operator completed in 26 usec.
Ctl: Set EncryptionEnforcement to On.
Warning. Request URI matched "[a-zA-Z0-9]" at REQUEST_URI.
Encryption parameter = [aafb509403bbf7d78c3c8fe1735d49f01b90eb64]
 , uri = [13111af1153095e85c70f8877b9126124908a771] [file
"/usr/local/apache/conf/crs/base_rules/modsecurity_crs_15_custom.
conf"] [line "31"] [id "1000"] [msg "Request Validation
Violation."]
```

Missing Hash Token

If the attacker simply removes the `rv_token`, the rules warn on that attempt as well:

```
Rule 100909d20: SecRule "REQUEST_URI" "@validateEncryption
[a-zA-Z0-9]"
 "phase:2,log,id:1000,t:none,block,msg:'Request
Validation Violation.',ctl:encryptionEnforcement=On"
Transformation completed in 0 usec.
Executing operator "validateEncryption" with param "[a-zA-Z0-9]"
against REQUEST_URI.
Target value: "/wordpress/?p=4%27"
Request URI without encryption parameter [/wordpress/?p=4%27]
Operator completed in 13 usec.
Ctl: Set EncryptionEnforcement to On.
Warning. Request URI matched "[a-zA-Z0-9]" at REQUEST_URI. No

Encryption parameter [file "/usr/local/apache/conf/
crs/base_rules/modsecurity_crs_15_custom.conf"]
[line "31"] [id "1000"] [msg "Request Validation Violation."]
```

Recipes in Part II of this book show how this request validation token injection technique protects against other attack categories.

[3] `http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual`

RECIPE 1-3: INSTALLING THE OWASP MODSECURITY CORE RULE SET (CRS)

This recipe shows you how to install and quickly configure the web application attack detection rules from the OWASP ModSecurity CRS. When this book was written, the CRS version was 2.2.5. Note that the rule logic described in this recipe may change in future versions of the CRS.

Ingredients

- OWASP ModSecurity CRS version 2.2.5[4]
  - modsecurity_crs_10_setup.conf
  - modsecurity_crs_20_protocol_violations.conf
  - modsecurity_crs_21_protocol_anomalies.conf
  - modsecurity_crs_23_request_limits.conf
  - modsecurity_crs_30_http_policy.conf
  - modsecurity_crs_35_bad_robots.conf
  - modsecurity_crs_40_generic_attacks.conf
  - modsecurity_crs_41_sql_injection_attacks.conf
  - modsecurity_crs_41_xss_attacks.conf
  - modsecurity_crs_45_trojans.conf
  - modsecurity_crs_47_common_exceptions.conf
  - modsecurity_crs_49_inbound_blocking.conf
  - modsecurity_crs_50_outbound.conf
  - modsecurity_crs_59_outbound_blocking.conf
  - modsecurity_crs_60_correlation.conf

OWASP ModSecurity CRS Overview

ModSecurity, on its own, has no built-in protections. To become useful, it must be configured with rules. End users certainly can create rules for their own use. However, most users have neither the time nor the expertise to properly develop rules to protect themselves from emerging web application attack techniques. To help solve this problem, the Trustwave SpiderLabs Research Team developed the OWASP ModSecurity CRS. Unlike intrusion detection and prevention systems, which rely on signatures specific to known vulnerabilities, the CRS provides *generic attack payload detection* for unknown vulnerabilities often found in web applications. The advantage of this generic approach is that the CRS can protect both public software and custom-coded web applications.

Core Rules Content

To protect generic web applications, the Core Rules use the following techniques:

- **HTTP protection** detects violations of the HTTP protocol and a locally defined usage policy.
- **Real-time blacklist lookups** use third-party IP reputation.
- **Web-based malware detection** identifies malicious web content by checking against the Google Safe Browsing API.
- **HTTP denial-of-service protection** defends against HTTP flooding and slow HTTP DoS attacks.
- **Common web attack protection** detects common web application security attacks.

- **Automation detection** detects bots, crawlers, scanners, and other surface malicious activity.
- **Integration with AV scanning for file uploads** detects malicious files uploaded through the web application.
- **Tracking sensitive data** tracks credit card usage and blocks leakages.
- **Trojan protection** detects access to Trojan horses.
- **Identification of application defects** alerts on application misconfigurations.
- **Error detection and hiding** disguises error messages sent by the server.

Configuration Options

After you have downloaded and unpacked the CRS archive, you should edit the Apache httpd.conf file and add the following directives to activate the CRS files:

```
<IfModule security2_module>
   Include conf/crs/modsecurity_crs_10_setup.conf
   Include conf/crs/activated_rules/*.conf
</IfModule>
```

Before restarting Apache, you should review/update the new modsecurity_crs_10_setup.conf.example file. This is the central configuration file, which allows you to control how the CRS works. In this file, you can control the following related CRS topics:

- Mode of detection: Traditional versus 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)

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 an action to take. This change makes it easy for the user to adjust settings in `SecDefaultAction`; these are inherited by `SecRule`s. This is a good approach for using a third-party set of rules, because our goal is *detecting issues*, not telling the user *how to react*. We also removed the logging actions from the rules so that the user can control exactly which files he or she wants to send logs to.

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 the rule logic is *self-contained*. Just like HTTP itself, the individual rules are stateless. This means that no intelligence is shared between rules, and no rule has insight into any previous rule matches. The rule uses only its current, single-rule logic for detection. In this mode, if a rule triggers, it executes any disruptive/logging actions specified on the current rule.

If you want to run the CRS in Traditional mode, you can do so easily by verifying that the `SecDefaultAction` line in the modsecurity_crs_10_setup.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"
```

With this configuration, when a CRS rule matches, it is denied, and the alert data is logged to both the Apache error_log file and the ModSecurity audit log file. Here is a sample error_log message for a SQL Injection attack:

```
[Fri Feb 17 14:40:48 2012] [error] [client 192.168.1.103]
ModSecurity: Warning. Pattern match "(?i:\\\\bunion\\\\b.{1,100}?
\\\\bselect\\\\b)" at ARGS:h_id. [file "/usr/local/apache/conf/
crs/base_rules/modsecurity_crs_41_ sql_injection_attacks.conf"]
[line "318"] [id "959047"] [rev "2.2.3"] [msg "SQL Injection
Attack"] [data "uNiOn/**/sEleCt"] [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 "192.168.1.103"][uri "/index.php"]
[unique_id "Tz6tQMCoAWcAAIykJgYAAAAA
```

Pros and Cons of Traditional Detection Mode

Pros

- It's relatively easy for a new user to understand the detection logic.
- Better performance is possible (lower latency/resources), because the first disruptive match stops further processing.

Cons

- It's not optimal from a rules management perspective (handling false positives and implementing exceptions):
  - It's difficult to edit a rule's complex regular expressions. The typical method is to copy and paste the existing rule into a local custom rules file, edit the logic, and then disable the existing CRS rule. The end result is that heavily customized rule sets are not updated when new CRS versions are released.
- It's 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 several lower-severity alerts in aggregate could be.

Anomaly Scoring Detection Mode (Collaborative Rules Concept)

This advanced inspection mode implements the concepts of *collaborative detection* and *delayed blocking*. In this mode, the inspection and detection logic is decoupled from the blocking functionality within the rules. The individual rules can be run so that the detection remains. However, instead of applying any disruptive action at that point, the rules contribute to an overall *transactional anomaly score* collection. In addition, each rule stores metadata about each rule match (such as the rule ID, attack category, matched location, and matched data) within a unique temporary transactional (`TX`) variable.

If you want to run the CRS in Anomaly Scoring mode, you can do so easily by updating the `SecDefaultAction` line in the modsecurity_crs_10_setup.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 of operation, each matched rule does not block. Instead, it increments anomaly scores using ModSecurity's `setvar` action. Here is an example of the SQL Injection CRS rule that generated the previous alert. As you can see, the rule uses `setvar` actions to increase both the overall anomaly score and the SQL Injection subcategory score:

```
SecRule REQUEST_COOKIES|REQUEST_COOKIES_NAMES|REQUEST_FILENAME|
ARGS_NAMES|ARGS|XML:/* "(?i:\bunion\b.{1,100}?\bselect\b)" \
  "phase:2,rev:'2.2.3',capture,multiMatch,t:none,t:urlDecodeUni,\
t:replaceComments,ctl:auditLogParts=+E,block,\
msg:'SQL Injection Attack',id:'959047',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}"
```

Anomaly Scoring Severity Levels

Each rule has a `severity` level specified. The updated rules action dynamically increments the anomaly score value by using macro expansion. Here's an example:

```
SecRule REQUEST_COOKIES|REQUEST_COOKIES_NAMES|REQUEST_FILENAME|
ARGS_NAMES|ARGS|XML:/* "(?i:\bunion\b.{1,100}?\bselect\b)" \
  "phase:2,rev:'2.2.3',capture,multiMatch,t:none,t:urlDecodeUni,\
t:replaceComments,ctl:auditLogParts=+E,block,msg:'SQL Injection \
Attack',id:'959047',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}"
```

Using macro expansion allows the user to set his or her own anomaly score values from within the modsecurity_crs_10_setup.conf file. These values are propagated for use in the rules.

```
#
# -=[ Anomaly Scoring Severity Levels ]=-
#
# These are the default scoring points for each severity level.
# You may adjust these to your 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,id:'981207',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 means that every CRS rule that has a Severity rating of `critical` (such as `severity:'2'`) 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 "(?i:\\bunion\\b.{1,100}?\\
bselect\\b)" against ARGS:h_id.
Target value: "-50/**/uNiOn/**/sEleCt/**/1,2,3,4,5,6,7,8,9,10,1,2
,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3
,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4
,5,6,7,8,9/**/fRoM/**/jos_users--"
```

```
Added regex subexpression to TX.0: uNiOn/**/sEleCt
Operator completed in 33 usec.
Ctl: Set auditLogParts to ABIJDEFHE.
Setting variable: tx.msg=%{rule.msg}
Resolved macro %{rule.msg} to: SQL Injection Attack
Set variable "tx.msg" to "SQL Injection Attack".
Setting variable: tx.sql_injection_score=
+%{tx.critical_anomaly_score}
Original collection variable: tx.sql_injection_score = "6"
Resolved macro %{tx.critical_anomaly_score} to: 5
Relative change: sql_injection_score=6+5
Set variable "tx.sql_injection_score" to "11".
Setting variable: tx.anomaly_score=+%{tx.critical_anomaly_score}
Original collection variable: tx.anomaly_score = "8"
Resolved macro %{tx.critical_anomaly_score} to: 5
Relative change: anomaly_score=8+5
Set variable "tx.anomaly_score" to "13".
Setting variable: tx.%{rule.id}-WEB_ATTACK/SQL_INJECTION-
%{matched_var_name}=%{tx.0}Resolved macro %{rule.id} to: 959047
Resolved macro %{matched_var_name} to: ARGS:h_id
Resolved macro %{tx.0} to: uNiOn/**/sEleCtSet variable
"tx.959047-WEB_ATTACK/SQL_INJECTION-ARGS:h_id" to "uNiOn/**/
sEleCt".
Resolved macro %{TX.0} to: uNiOn/**/sEleCt
Warning. Pattern match "(?i:\\bunion\\b.{1,100}?\\bselect\\b)" at
 ARGS:h_id. [file "/usr/local/apache/conf/crs/base_rules/
modsecurity_crs_41_sql_injection_attacks.conf"]
[line "318"] [id "959047"] [rev "2.2.3"] [msg "SQL Injection
Attack"] [data "uNiOn/**/sEleCt"] [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"]
```

Anomaly Scoring Threshold Levels (Blocking)

Now that we can do anomaly scoring, the next step is to set our thresholds. If the current transactional score is above this score value, it is denied. Two different anomaly scoring thresholds must be set. One is set for the inbound request, which is evaluated at the end of `phase:2` in the modsecurity_crs_49_inbound_blocking.conf file. Another is set for outbound information leakages, which are 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,id:'981208',t:none,nolog,pass,\
setvar:tx.inbound_anomaly_score_level=5"
SecAction "phase:1,id:'981209',t:none,nolog,pass,\
setvar:tx.outbound_anomaly_score_level=4"
```

With these settings, Anomaly Scoring mode acts much like Traditional mode from a blocking perspective. Because all critical-level rules increase the anomaly score by 5 points, the end result is that even one critical-level rule match causes a block. If you want to adjust the anomaly score so that you have a lesser chance of blocking nonmalicious 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 match before you decide to block. Another advantage of this approach is that you can aggregate multiple lower-severity rule matches and then decide to block. 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, the request would be blocked.

Enable/Disable Blocking

The `SecRuleEngine` directive allows you to globally control blocking mode (`On`) versus Detection mode (`DetectionOnly`). With the new Anomaly Scoring Detection mode, if you want to allow blocking, you should set `SecRuleEngine On` and then set the following `TX` variable in the modsecurity_crs_10_setup.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,id:'981206',t:none,nolog,pass,\
setvar:tx.anomaly_score_blocking=on"
```

Now that this variable is set, the rule within the modsecurity_crs_49_inbound_blocking .conf file evaluates the anomaly scores at the end of the request phase and blocks the request:

```
# Alert and Block based on Anomaly Scores
#
SecRule TX:ANOMALY_SCORE "@gt 0" \
    "chain,phase:2,id:'981176',t:none,deny,log,msg:'Inbound \
Anomaly Score Exceeded (Total Score: %{TX.ANOMALY_SCORE},\
SQLi=%{TX.SQL_INJECTION_SCORE}, XSS=%{TX.XSS_SCORE}): Last \
Matched Message: %{tx.msg}',logdata:'Last Matched Data: \
%{matched_var}',setvar:tx.inbound_tx_msg=%{tx.msg},\
setvar:tx.inbound_anomaly_score=%{tx.anomaly_score}"
        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 another rule is commented out by default. This sample rule shows how you could alternatively choose to inspect/block based on a subcategory anomaly score (in this example for SQL Injection).

Alert Management: Correlated Events

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

To achieve this capability, the CRS can be run in a correlated event mode. Each individual rule generates a modsec_audit.log event Message entry but does not log to the error_log on its own. These rules are considered *basic* or *reference events* that have contributed to the overall anomaly score. They 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_setup.conf file:

```
#
# -=[ 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 log the standard Message data to the modsec_audit .log file. You receive only one 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 looks like this:

```
[Fri Feb 17 15:55:16 2012] [error] [client 192.168.1.103]
ModSecurity: Warning. Pattern match "(.*)" at TX:0. [file
"/usr/local/apache/conf/crs/base_rules/
modsecurity_crs_49_inbound_blocking.conf"] [line "26"]
```

```
[id "981176"] [msg "Inbound Anomaly Score Exceeded (Total Score:
 78, SQLi=28, XSS=): Last Matched Message: 981247-Detects concat
enated basic SQL injection and SQLLFI attempts"] [data "Last
Matched Data: -50 uNiOn"] [hostname "192.168.1.103"] [uri
"/index.php"] [unique_id "Tz6@tMCoqAEAAM5lMk0AAAAA"]
```

This entry tells us that a SQL Injection attack was identified on the inbound request. We see that the total anomaly score is 78 and that the subcategory score of SQLi is 28. This tells us that a number of SQL Injection rules were triggered. If you want to see the details of all the reference events (individual rules that contributed to this correlated event), you can review the modsec_audit.log data for this transaction.

Pros and Cons of Anomaly Scoring Detection Mode

Pros

- Increased confidence in blocking. Because more detection rules contribute to the anomaly score, the higher the score, the more confidence you can have in blocking malicious transactions.
- It allows users to set a threshold that is appropriate for their site. Different sites may have different thresholds for blocking.
- It allows several low-severity events to trigger alerts while individual ones are suppressed.
- One correlated event helps alert management.
- Exceptions may be handled easily by increasing the overall anomaly score threshold.

Cons

- It's more complex for the average user.
- Log monitoring scripts may need to be updated for proper analysis.

Alert Management: Inbound/Outbound Correlation

One important alert management issue for security analysts to deal with is *prioritization*. From an incident response perspective, many ModSecurity/CRS users have a difficult time figuring out which alerts they need to fully review and follow up on. This is especially true if you're running ModSecurity in `DetectionOnly` mode, because you may get alerts, but you are not actively blocking attacks or information leakages.

If you are running the OWASP ModSecurity CRS in Anomaly Scoring mode, you have the added advantage of correlating rule matches to gather more intelligence about transactional issues.

The highest severity rating that an identified inbound attack can have is 2 (critical). To have a higher severity rating (1 or 0), you need to use correlation. At the end of both the request and response phases, the CRS saves the final rule match message data.

After the transaction has completed (in `phase:5` logging), the rules in the base_rules/modsecurity_crs_60_correlation.conf file conduct further postprocessing by analyzing any inbound events with any outbound events to provide a more intelligent, priority-based correlated event. Consider the following questions that security analysts typically need to answer when investigating web alerts:

- Did an inbound attack occur?
- Did an HTTP response status code error (4*xx*/5*xx* level) occur?
- Did an application information leakage event occur?

If an inbound attack was detected, and either an outbound application status code error or information leakage event was detected, the overall event severity is raised to one of the following:

- **0, EMERGENCY**, is generated from correlation of anomaly scoring data where an inbound attack and an outbound leakage exist.
- **1, ALERT**, is generated from correlation where an inbound attack and an outbound application-level error exist.

Sample Correlated Event

Let's look at a sample SQL Injection attack scenario. If an attacker sends inbound SQL Injection attack payloads and the application responds normally, you would see a normal inbound event. Although this is certainly useful information, this indicates only that an attacker has sent an attack. On the other hand, if the target application does not properly handle this input and instead returns technical information leakage data such as that shown in Figure 1-4, you would want to follow up on that issue and initiate blocking.

In this situation, the CRS correlates the inbound SQL Injection attack with the outbound application error/code leakage event and thus generates a higher-level severity alert message such as the following:

```
[Fri Feb 17 16:26:37 2012] [error] [client 192.168.1.103]
ModSecurity: Warning. Operator GE matched 1 at TX. [file "/usr/
local/apache/conf/crs/base_rules/modsecurity_crs_60_correlation.
conf"] [line "29"] [id "981202"] [msg "Correlated Attack Attempt
Identified: (Total Score: 21, SQLi=5, XSS=) Inbound Attack
(981242-Detects classic SQL injection probings 1/2 Inbound
Anomaly Score: 13) + Outbound Application Error (ASP/JSP source
code leakage - Outbound Anomaly Score: 8)"] [severity "ALERT"]
[hostname "192.168.1.103"] [uri "/zapwave/active/inject/inject-
sql-form-basic.jsp"] [unique_id "Tz7GDcCoAWcAAOsFI@cAAAAA"]
```

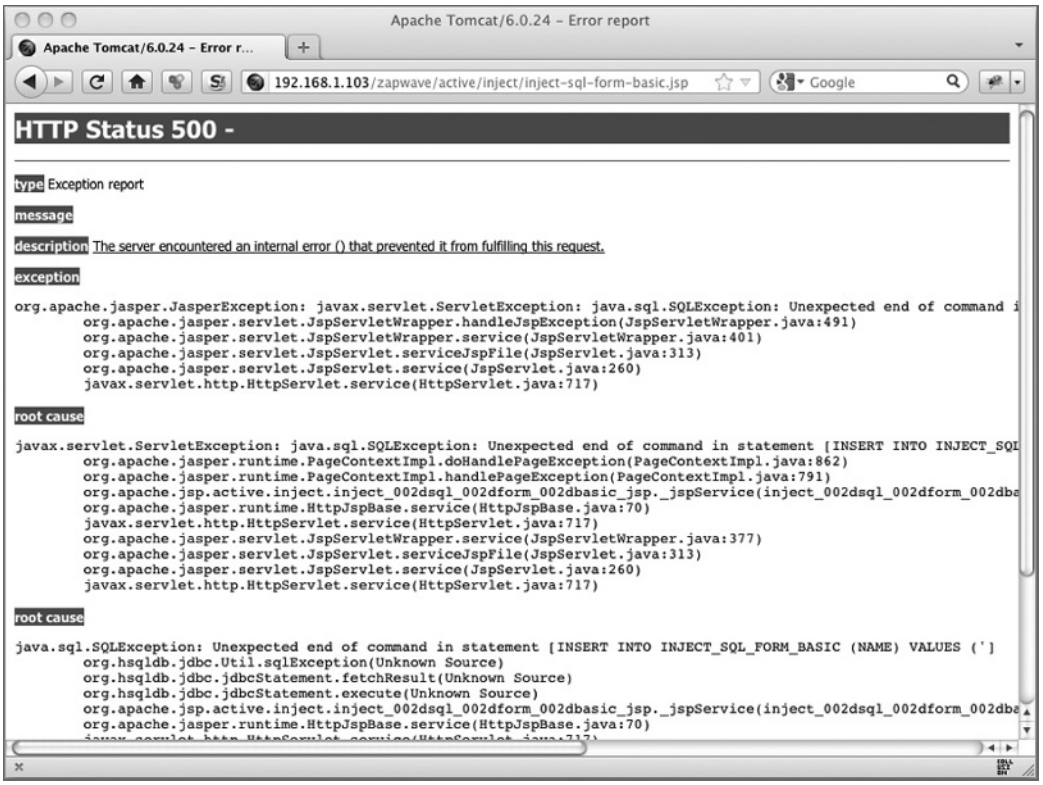

Figure 1-4: Sample Java stack dump response page

This correlated event provides much more actionable data to security analysts. It also allows them to implement more aggressive blocking mechanisms such as blocking all categories of SQL Injection attacks or blocking for only this particular parameter on the page. Correlated event analysis helps to expedite the incident response process and allows security operations teams to focus their efforts on actionable situations instead of only data from inbound attacks.

[4]https://www.owasp.org/index.php/
Category:OWASP_ModSecurity_Core_Rule_Set_Project

RECIPE 1-4: INTEGRATING INTRUSION DETECTION SYSTEM SIGNATURES

This recipe shows you how to integrate public Snort IDS web attack signatures within ModSecurity.

Ingredients

- OWASP ModSecurity CRS[5]
- Emerging Threats (ET) Snort Rules (for Snort v2.8.4)[6]
  - emerging-web_server.rules
  - emerging-web_specific_apps.rules

Emerging Threats' Snort Web Attack Rules

You may be familiar with the Emerging Threats project. It has a few Snort rules files related to known web application vulnerabilities and attacks:

- emerging-web_server.rules
- emerging-web_specific_apps.rules

Here is a sample ET rule taken from the emerging-web_specific_apps.rules file that describes a known SQL Injection vulnerability in the 20/20 Auto Gallery application:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS  (msg:"ET
 WEB_SPECIFIC_APPS 20/20 Auto Gallery SQL Injection Attempt –
vehiclelistings.asp vehicleID SELECT";  flow:established,to_server
; uricontent:"/vehiclelistings.asp?"; nocase;  uricontent:
"vehicleID="; nocase; uricontent:"SELECT"; nocase;  pcre:"/.
+SELECT.+FROM/Ui"; classtype:web-application-attack; reference:cve
,CVE-2006-6092; reference:url,www.securityfocus.com/bid/21154;
  reference:url,doc.emergingthreats.net/2007504;
  reference:url,www.emergingthreats.net/cgi-bin/cvsweb.cgi/sigs/
WEB_SPECIFIC_APPS/WEB_2020_Auto_gallery; sid:2007504; rev:5;)
```

When reviewing this web attack rule, we can conclude that there is a SQL Injection vulnerability in the /vehiclelistings.asp page, presumably in the `vehicle` parameter payload. This tells us *where* the injection point is located within the web request. A regular expression check then looks for specific SQL values. This data tells us *what* data it is looking for to detect an attack. Upon deeper analysis, however, we find a few accuracy concerns:

- The injection point accuracy is not ideal, because the rule uses the older Snort `uricontent` keyword. What happens if the vehiclelistings.asp page also accepts parameter data within `POST` payloads? This would mean that the `vehicleID` parameter might actually be passed in the request body and not in the `QUERY_STRING`. This occurrence would result in a false negative, and the rule would not match.

- The regular expression analysis is not constrained to only the `vehicleID` parameter data. The rule triggers if these three pieces of data *exist* in the request stream and not if the regular expression match is found *only* within the `vehicleID` parameter payload.
- The regular expression is not comprehensive, because it looks for only a small subset of possible SQL Injection attack data. Many other types of SQL Injection attack payloads would bypass this basic check. The result is that the Snort rule writers would have to create many copies of this rule, each with different regular expression checks.

The Value of Attacks against Known Vulnerabilities

As opposed to the generic attack payload detection used by the OWASP ModSecurity CRS, these ET Snort rules are developed based on vulnerability information for public software. Identifying attacks against known vulnerabilities does have value in the following scenarios:

- If your organization is using the targeted application, it can raise the threat level, lessen false positives, and ultimately provide increased confidence in blocking.
- Even if you are not running the targeted software in your enterprise, you still might want to be made aware of attempts to exploit known vulnerabilities, regardless of their chances of success.

To summarize, *the value of these signatures lies in identifying a known attack vector location (injection point)*. We can leverage this data in the CRS by converting the ET Snort rule into a ModSecurity rule and correlating the information with anomaly scoring.

Using Anomaly Scoring Mode with the CRS

Recipe 1-3 outlined how to run the CRS in either Traditional or Anomaly Scoring mode. The main benefit of anomaly scoring is increased intelligence. Not only can more rules contribute to an anomaly score, but each rule also saves valuable metadata about rule matches in temporary `TX` variables. Let's look at CRS rule ID 959047 as an example:

```
SecRule REQUEST_COOKIES|REQUEST_COOKIES_NAMES|REQUEST_FILENAME|
ARGS_NAMES|ARGS|XML:/* "(?i:\bunion\b.{1,100}?\bselect\b)" \
 "phase:2,rev:'2.2.3',capture,multiMatch,t:none,t:urlDecodeUni
,t:replaceComments,ctl:auditLogParts=+E,block,msg:'SQL
Injection Attack',id:'959047',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 bold `setvar` action is the key piece of data to understand. If a rule matches, we initiate a `TX` variable that contains metadata about the match:

- `tx.%{rule.id}` uses macro expansion to capture the rule ID value data and saves it in the `TX` variable name.
- `WEB_ATTACK/SQL_INJECTION` captures the attack category data and saves it in the `TX` variable name.
- `%{matched_var_name}` captures the variable location of the rule match and saves it in the `TX` variable name.
- `%{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 processes a sample request, we see the following:

```
Executing operator "rx" with param "(?i:\\bunion\\b.{1,100}?\\b
select\\b)" against ARGS:vehicleID.
Target value: "9999999/**/union/**/select/**/0,0,0,0,0,0,0,0,0,
0,0,0,0x33633273366962,0,0,0,0,0,0,0/**/from/**/jos_users--"
Added regex subexpression to TX.0: union/**/select
Operator completed in 18 usec.
Ctl: Set auditLogParts to ABIJDEFHE.
Setting variable: tx.msg=%{rule.msg}
Resolved macro %{rule.msg} to: SQL Injection Attack
Set variable "tx.msg" to "SQL Injection Attack".
Setting variable: tx.sql_injection_score=+%{tx.critical_anomaly_
score}
Original collection variable: tx.sql_injection_score = "10"
Resolved macro %{tx.critical_anomaly_score} to: 5
Relative change: sql_injection_score=10+5
Set variable "tx.sql_injection_score" to "15".
Setting variable: tx.anomaly_score=+%{tx.critical_anomaly_score}
Original collection variable: tx.anomaly_score = "13"
Resolved macro %{tx.critical_anomaly_score} to: 5
Relative change: anomaly_score=13+5
Set variable "tx.anomaly_score" to "18".
Setting variable: tx.%{rule.id}-WEB_ATTACK/SQL_INJECTION-
%{matched_var_name}=
%{tx.0}
Resolved macro %{rule.id} to: 959047
Resolved macro %{matched_var_name} to: ARGS:vehicleID
Resolved macro %{tx.0} to: union/**/select
Set variable "tx.959047-WEB_ATTACK/SQL_INJECTION-ARGS:vehicleID"
 to "union/**/select".
```

The final, bold entry shows the `TX` variable data that is now at our disposal. This `TX` data tells us that a SQL Injection attack payload was detected in a parameter called `vehicleID`. We can now use this type of data to correlate with converted Snort attack signatures.

Converting Snort Signatures in ModSecurity Rule Language

We can convert the Snort rule just discussed into a ModSecurity rule like this:

```
# (2007545) SpiderLabs Research (SLR) Public Vulns:
# ET WEB_SPECIFIC_APPS 20/20 Auto Gallery SQL Injection Attempt -
# vehiclelistings.asp vehicleID UPDATE
SecRule REQUEST_LINE "@contains /vehiclelistings.asp" "chain,
phase:2,block,t:none,
t:urlDecodeUni,t:htmlEntityDecode,t:normalisePathWin,capture,
nolog,auditlog,logdata:'%{TX.0}',severity:'2',id:2007545,rev:6,
 msg:'SLR: ET WEB_SPECIFIC_APPS 20/20 Auto Gallery SQL
Injection Attempt -- vehiclelistings.asp vehicleID UPDATE',
tag:'web-application-attack',tag:'url,www.securityfocus.com/bid
/21154'"

      SecRule TX:'/WEB_ATTACK/SQL_INJECTION.*ARGS:vehicleID/' ".*"
 "capture,ctl:auditLogParts=+E,setvar:'tx.msg=%{tx.msg} - ET WEB_
SPECIFIC_APPS 20/20 Auto Gallery SQL Injection Attempt - vehicle
listings.asp vehicleID UPDATE',setvar:tx.anomaly_score=+20,
setvar:'tx.%{rule.id}-WEB_ATTACK-%{rule.severity}-
%{rule.msg}-%{matched_var_name}=%{matched_var}'"
```

As you can see, the first `SecRule` checks the request line data to make sure that it matches the vulnerable resource. We then run a second chained rule that, instead of looking separately for the existence of the parameter name and a regular expression check, simply inspects previously matched `TX` variable metadata. In this case, if a previous CRS rule identified a SQL Injection attack payload in the `ARGS:vehicleID` parameter location, the rule matches. Here is how the final rule processing looks in the debug log. You can see that we find a match of previously generated SQL Injection event data found within the vulnerable parameter location from the Snort ET signature:

```
Recipe: Invoking rule 10217f828; [file "/usr/local/apache/conf/
crs/base_rules/modsecurity_crs_46_slr_et_sqli_attacks.conf"]
[line "52"].
Rule 10217f828: SecRule "TX:'/SQL_INJECTION.*ARGS:vehicleID/'"
"@rx .*" "capture,ctl:auditLogParts=+E,setvar:'tx.msg=ET WEB_
SPECIFIC_APPS 20/20 Auto Gallery SQL Injection Attempt -
vehiclelistings.asp vehicleID UPDATE',
setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},
setvar:tx.%{rule.id}-WEB_ATTACK/SQL_INJECTION-%{matched_var_
name}=%{matched_var}"
Expanded "TX:'/SQL_INJECTION.*ARGS:vehicleID/'" to "TX:981260-
WEB_ATTACK/SQL_INJECTION-ARGS:vehicleID|TX:981231-WEB_ATTACK/
SQL_INJECTION-ARGS:vehicleID|TX:959047-WEB_ATTACK/SQL_INJECTION
-ARGS:vehicleID|TX:959073-WEB_ATTACK/SQL_INJECTION-ARGS:
vehicleID".
Transformation completed in 0 usec.
Executing operator "rx" with param ".*" against TX:981260-WEB_
ATTACK/SQL_INJECTION-ARGS:vehicleID.
```

```
Target value: ",0x33633273366962"
Added regex subexpression to TX.0: ,0x33633273366962
Operator completed in 19 usec.
Ctl: Set auditLogParts to ABIJDEFHEEEEE.
Setting variable: tx.msg=ET WEB_SPECIFIC_APPS 20/20 Auto
Gallery SQL Injection Attempt -- vehiclelistings.asp vehicleID
UPDATE
Set variable "tx.msg" to "ET WEB_SPECIFIC_APPS 20/20 Auto Gallery
 SQL Injection Attempt -- vehiclelistings.asp vehicleID UPDATE".
Setting variable: tx.anomaly_score=+%{tx.critical_anomaly_score}
Original collection variable: tx.anomaly_score = "83"
Resolved macro %{tx.critical_anomaly_score} to: 5
Relative change: anomaly_score=83+5
Set variable "tx.anomaly_score" to "88".
Setting variable: tx.%{rule.id}-WEB_ATTACK/SQL_INJECTION-
%{matched_var_name}=%{matched_var}
Resolved macro %{rule.id} to: 2007545
Resolved macro %{matched_var_name} to: TX:981260-WEB_ATTACK/
SQL_INJECTION-ARGS:vehicleID
Resolved macro %{matched_var} to: ,0x33633273366962
Set variable "tx.2007545-WEB_ATTACK/SQL_INJECTION-TX:981260-
WEB_ATTACK/SQL_INJECTION-ARGS:vehicleID" to ",0x33633273366962".
Resolved macro %{TX.0} to: ,0x33633273366962
Resolved macro %{TX.0} to: ,0x33633273366962
Warning. Pattern match ".*" at TX:981260-WEB_ATTACK/SQL_INJECTION
-ARGS:vehicleID. [file "/usr/local/apache/conf/crs/base_rules/
modsecurity_crs_46_slr_et_sqli_attacks.conf"] [line "51"]
[id "2007545"] [rev "6"] [msg "SLR: ET WEB_SPECIFIC_APPS 20/20
Auto Gallery SQL Injection Attempt -- vehiclelistings.asp
vehicleID UPDATE"] [data ",0x33633273366962"]
[severity "CRITICAL"] [tag "web-application-attack"] [tag
 "url,www.securityfocus.com/bid/21154"]
```

By combining the generic attack payload detection of the OWASP ModSecurity CRS with the specific known attack vectors from the Snort ET web signatures, we can more accurately identify malicious requests and apply more aggressive response actions.

To make the conversion of the ET Snort web attack rules easier, the Trustwave SpiderLabs Research Team has included a Perl script in the OWASP ModSecurity CRS distribution called snort2modsec2.pl that autoconverts the rules for you. Here is an example of running the script and viewing a sample of the output:

```
$ ./snort2modsec2.pl emerging-web_specific_apps.rules >
modsecurity_crs_46_snort_attacks.conf
$ head -6 modsecurity_crs_46_snort_attacks.conf
SecRule REQUEST_FILENAME "!@pmFromFile snort2modsec2_static.data"
 "phase:2,
nolog,pass,t:none,t:urlDecodeUni,t:htmlEntityDecode,t:normalise
PathWin,skipAfter:END_SNORT_RULES"
```

```
# (sid 2011214) ET WEB_SPECIFIC_APPS ArdeaCore pathForArdeaCore
Parameter Remote File Inclusion Attempt
SecRule REQUEST_URI_RAW "(?i:\/ardeaCore\/lib\/core\/ardeaInit\.
php)" "chain,
phase:2,block,t:none,t:urlDecodeUni,t:htmlEntityDecode,t:normalise
PathWin,capture,
nolog,auditlog,logdata:'%{TX.0}',id:sid2011214,rev:2,msg:'ET WEB_
SPECIFIC_APPS
 ArdeaCore pathForArdeaCore Parameter Remote File Inclusion
Attempt',
tag:'web-application-attack',tag:'url,doc.emergingthreats.net/
2011214'"
SecRule REQUEST_URI_RAW "@contains GET " "chain"
SecRule ARGS:pathForArdeaCore "(?i:\s*(ftps?|https?|php)\:\/)"
 "ctl:auditLogParts=+E,setvar:'tx.msg=ET WEB_SPECIFIC_APPS
ArdeaCore
 pathForArdeaCore Parameter Remote File Inclusion Attempt',
setvar:tx.anomaly_score=+20,setvar:'tx.%{rule.id}-WEB_ATTACK-
%{matched_var_name}=%{matched_var}'"
```

[5] https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project

[6] http://rules.emergingthreats.net/open/snort-2.8.4/rules/

RECIPE 1-5: USING BAYESIAN ATTACK PAYLOAD DETECTION

This recipe shows you how to integrate Bayesian analysis of HTTP parameter payloads to identify malicious data.

Ingredients

- ModSecurity Reference Manual[7]
  - Lua API
- OWASP ModSecurity CRS Lua scripts[8]
  - modsecurity_crs_48_bayes_analysis.conf
  - bayes_train_spam.lua
  - bayes_train_ham.lua
  - bayes_check_spam.lua
- OSBF-Lua: Bayesian text classifier[9]
- Moonfilter: Lua wrapper for OSBF-Lua[10]
- Moonrunner: Command-line interface to Moonfilter[11]

Using Bayesian Analysis to Detect Web Attacks

Bayesian text classifiers have long been used to detect spam e-mails. Why not use the same type of analysis for web traffic to identify malicious requests? The general concept is directly applicable, from e-mail analysis to HTTP request parameter inspection. But there are a few nuances to be aware of:

- **Ham versus spam.** In our implementation, *ham* is considered *nonmalicious traffic*, and *spam* is considered an *attack payload*.
- **Input source.** Bayesian classifiers normally inspect operating system (OS) text files (e-mail messages) with many different lines of text. With this proof-of-concept implementation in ModSecurity, we must bypass feeding the Bayesian classifier data from OS text files, because this would incur too much latency. We instead must pass the data directly from the request to the Bayesian classifier using the Lua API and store the text in temporary variables.
- **Data format.** E-mail messages have a certain format and construction, with MIME headers at the top and then the body of the message. This format can impact the overall scores of the classifiers. In our current implementation with ModSecurity, however, we pass only text payloads from individual parameters. This smaller dataset may impact the final classifications.

The information presented within this recipe is not meant to be a primer on the inner workings of Bayesian classifier theory. Instead, it is a practical proof-of-concept implementation using ModSecurity's Lua API. If you would like more technical information on the algorithms used with the Bayesian classification, I suggest you read Paul Graham's seminal paper, titled "A Plan for Spam,"[12] and his follow-up, "Better Bayesian Filtering."[13]

OSBF-Lua Installation

Bayesian classification is implemented into ModSecurity through its Lua API. The first component to install is a Lua module created by Fidelis Assis called OSBF-Lua. OSBF-Lua requires Lua 5.1 to be installed, with dynamic loading enabled. Follow these basic installation steps:

```
$ tar xvzf osbf-lua-x.y.z.tar.gz
$ cd osbf-lua-x.y.z
```

Before you compile, you should add the following patch to the osbf_bayes.c file. It fixes an overflow bug that may be encountered after extended training:

```
+++ osbf_bayes.c 2008-12-17 10:36:18.000000000 -0200
@@ -854,9 +854,9 @@
          if (cfx > 1)
          cfx = 1;
          confidence_factor = cfx *
```

```
-         pow ((diff_hits * diff_hits - K1 /
+         pow (((double)diff_hits * diff_hits - K1 /
               (class[i_max_p].hits + class[i_min_p].hits)) /
-              (sum_hits * sum_hits), 2) /
+             ((double)sum_hits * sum_hits), 2) /
          (1.0 +
          K3 / ((class[i_max_p].hits + class[i_min_p].hits) *
               feature_weight[window_idx]));
```

Then complete the compilation and installation with this:

```
$ make
$ make install
```

After OSBF-Lua is installed, the next step is to install Moonfilter.

Moonfilter Installation

Moonfilter, written by Christian Siefkes, is a wrapper script for OSBF-Lua that provides an easy interface for training and classification. After downloading the moonfilter.lua script, you should edit the file and update the final (bold) line of the Configuration section to look like this:

```
----- Exported configuration variables ----------------------------

-- Minimum absolute pR a correct classification must get not to
-- trigger a reinforcement.
threshold = 20
-- Number of buckets in the database. The minimum value
-- recommended for production is 94321.
buckets = 94321
-- Maximum text size, 0 means full document (default). A
-- reasonable value might be 500000 (half a megabyte).
max_text_size = 0
-- Minimum probability ratio over the classes a feature must have
-- not to be ignored. 1 means ignore nothing (default).
min_p_ratio = 1
-- Token delimiters, in addition to whitespace. None by default,
-- could be set e.g. to ".@:/".
delimiters = ""
-- Whether text should be wrapped around (by re-appending the
-- first 4 tokens after the last).
wrap_around = true
-- The directory where class database files are stored. Defaults
-- to the current working directory (empty string). Note that the
-- directory name MUST end in a path separator (typically '/' or
-- '\', depending on your OS) in all other cases. Changing this
-- value will only affect future calls to the |classes| command;
-- it won't change the location of currently active classes.
classdir = ""
```

```
-- The text to classify/train as a string -- can be set explictly
-- if desired
text = nil
```

The original setting is `local text = nil`. We must remove the word `local` so that Moonfilter allows us to set the classification text from within our own Lua scripts that will pass data dynamically directly from inbound HTTP requests.

Moonrunner Installation and Usage

Moonrunner, also by Christian Siefkes, is a command-line Lua script that you can use to manage the spam and ham database files and also conduct individual classification actions. After downloading Moonrunner, you should execute the following commands:

```
# ./moonrunner.lua
classes /var/log/httpd/spam /var/log/httpd/ham
classes ok
create
create ok
stats /var/log/httpd/spam
stats ok: "-- Statistics for /var/log/httpd/spam.cfc\
Database version:                    OSBF-Bayes\
Total buckets in database:                94321\
Buckets used (%):                           0.0\
Trainings:                                    0\
Bucket size (bytes):                         12\
Header size (bytes):                       4092\
Number of chains:                             0\
Max chain len (buckets):                      0\
Average chain length (buckets):               0\
\
"
readuntil <EOF>
12'UNION/*!00909SELECT 1,2,3,4,5,6,7,8,9 --
<EOF>
readuntil ok
train /var/log/httpd/spam
Invoking classify for ''
train ok: misclassified=false reinforced=true
stats /var/log/httpd/spam
stats ok: "-- Statistics for /var/log/httpd/spam.cfc\
Database version:                    OSBF-Bayes\
Total buckets in database:                94321\
Buckets used (%):                           0.0\
Trainings:                                    1\
Bucket size (bytes):                         12\
Header size (bytes):                       4092\
Number of chains:                            32\
Max chain len (buckets):                      1\
```

```
Average chain length (buckets):              1\
```

In this section of commands, we perform the following tasks:

1. Specify our two classification files (spam/ham).
2. Create the database for each classification.
3. Execute the `stats` command to see general statistics about the newly created spam database.
4. Specify a sample SQL Injection text string for training.
5. Train the classifier that the sample text was to be classified as spam.
6. Re-execute the `stats` command to see the updated information in the spam database.

This same approach also can, and should, be used to classify nonmalicious (ham) payloads:

```
readuntil <EOF>
this is just normal text.
<EOF>
readuntil ok
train /var/log/httpd/ham
Reusing stored result for ''
train ok: misclassified=true reinforced=false
stats /var/log/httpd/ham
stats ok: "-- Statistics for /var/log/httpd/ham.cfc\
Database version:                    OSBF-Bayes\
Total buckets in database:                94321\
Buckets used (%):                           0.0\
Trainings:                                    1\
Bucket size (bytes):                         12\
Header size (bytes):                       4092\
Number of chains:                            43\
Max chain len (buckets):                      1\
Average chain length (buckets):               1\
```

The next logical step is to submit a new string of text. Instead of training the classifier on it, we try to classify it as either spam or ham:

```
readuntil <EOF>
1'UNION/*!0SELECT user,2,3,4,5,6,7,8,9/*!0from/*!0mysql.user/*-
<EOF>
readuntil ok
classify
classify ok: prob=0.5 probs=[ 0.5 0.5 ] class=/var/log/httpd/spam
 pR=0 reinforce=true
train /var/log/httpd/spam
```

```
Reusing stored result for ''
train ok: misclassified=true reinforced=false
classify
classify ok: prob=0.73998695843754 probs=[ 0.73998695843754
0.26001304156246 ] class=/var/log/httpd/spam pR=0.26799507117831
 reinforce=true
```

This SQL Injection payload was correctly classified as spam, but the probability was only 0.5. The closer the score comes to 1.0, the more confident the classifier is of the classification. We then train the classifier as spam, and the new classification score probability is 0.73998695843754.

Ongoing Moonrunner Usage

Moonrunner is a useful tool after you have deployed the ModSecurity component in production. Moonrunner allows you to periodically run stats checks to verify the trainings of the two classifier databases. You can also use Moonrunner to manually retrain payloads taken from audit log data if ModSecurity improperly flagged them.

The Advantage of Bayesian Analysis

The ModSecurity OWASP CRS, like most security systems, relies heavily on the use of blacklist regular expression filters to identify malicious payloads. Although this approach provides a base level of protection, it offers insufficient protection against a determined attacker. The main shortcoming of using regular expressions for attack detection is that the operator check's result is binary: It either matches, or it doesn't. There is no middle ground. This means that an attacker may run through an iterative process of trial and error, submitting attack payloads until he or she finds a permutation that bypasses the regular expression logic. Let's take a quick look at a sample evasion for one of the SQL Injection rules presented earlier:

```
SecRule REQUEST_COOKIES|REQUEST_COOKIES_NAMES|REQUEST_FILENAME|
ARGS_NAMES|ARGS|
XML:/* "(?i:\bunion\b.{1,100}?\bselect\b)" \
     "phase:2,rev:'2.2.3',capture,multiMatch,t:none,
t:urlDecodeUni,t:replaceComments,ctl:auditLogParts=+E,block,
msg:'SQL Injection Attack',id:'959047',
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 bold regular expression basically means that we are doing a case-insensitive search for the words "union" and "select" within 100 characters of each other. When an attacker sends in his or her initial attack probes, such as the following examples taken from the ModSecurity SQL Injection Challenge, they are all caught until the final evasion payload, shown in bold:

```
div 1 union%23%0Aselect 1,2,current_user
div 1 union%23foo*/*bar%0Aselect 1,2,current_user
div 1 union%23foofoofoofoo*/*bar%0Aselect 1,2,current_user
div 1 union%23foofoofoofoofoofoofoofoofoofoo*/*bar%0Aselect 1,2,
current_user
...
div 1 union%23foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo
foofoofoofoo
foofoofoofoofoofoofoofoofoofoofoofoofoo*/*bar%0Aselect 1,2,current_
user
```

The final payload evades the regular expression logic by padding the space between the `union` and `select` keywords with SQL comment text that the SQL database ignores. The final payload is also functionally equivalent to all the ones before it while bypassing the regular expression logic. Keep in mind, however, that the blacklist signatures do in fact work, for a period of time, and provide some level of hacking resistance. Using Bayesian analysis combined with blacklist regular expression inspection has two advantages:

- We can use the blacklist filters to identify the initial attack attempts and use the payloads that they identify to actually train the Bayesian classifiers that the payload is spam. So, in effect, the attackers train our detection logic. Remember that the final attack payload that can bypass a regular expression check is actually very similar to the previous versions that were detected. Usually, it comes down to a difference of only one character.
- Rather than a binary result, Bayesian analysis gives us a *probability* that a payload is malicious. With this approach, we now have a wider scale with which to identify the likelihood that a payload is bad.

With the Bayesian analysis in place, the final SQL Injection payload that evaded the ModSecurity `SecRule` filter is detected:

```
readuntil <EOF>
div 1 union%23foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo
foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo*/*bar%0Aselect 1
,2,current_user
<EOF>
readuntil ok
classify
classify ok: prob=0.99999999973866 probs=[ 0.99999999973866 2.613
```

```
4432207688e-10 ] class=/var/log/httpd/spam pR=5.6538442891482
reinforce=true
```

Integrating Bayesian Analysis with ModSecurity

With these components in place, the next step is to hook the Bayesian analysis components into ModSecurity so that the training and classification data comes directly from live application users. The first step in this process is to ensure that the ham and spam database files have read/write permission for the Apache user. Execute the following commands to change the ownership to the Apache user:

```
# ls -l *.cfc
-rw------- 1 root root 1135948 Feb 18 14:42 ham.cfc
-rw------- 1 root root 1135948 Feb 18 14:43 spam.cfc
# chown apache:apache *.cfc
# ls -l *.cfc
-rw------- 1 apache apache 1135948 Feb 18 14:42 ham.cfc
-rw------- 1 apache apache 1135948 Feb 18 14:43 spam.cfc
```

The next step is to activate the modsecurity_crs_48_bayes_analysis.conf file by adding it to your activated rules. Here are the contents of the rules file:

```
SecRule TX:'/^\\\d.*WEB_ATTACK/' ".*" "phase:2,t:none,log,pass,
logdata:'%{tx.bayes_msg}',exec:/etc/httpd/modsecurity.d/bayes_
train_spam.lua"

#SecRuleScript /etc/httpd/modsecurity.d/bayes_check_spam.lua
"phase:2,t:none,block,msg:'Bayesian Analysis Detects Probable
Attack.',logdata:'Score: %{tx.bayes_score}',severity:'2',
tag:'WEB_ATTACK/SQL_INJECTION',tag:'WASCTC/WASC-19',
tag:'OWASP_TOP_10/A1',tag:'OWASP_AppSensor/CIE1',
tag:'PCI/6.5.2',setvar:'tx.msg=%{rule.msg}',
setvar:tx.anomaly_score=+%{tx.critical_anomaly_score},
setvar:tx.%{rule.id}-WEB_ATTACK/BAYESIAN-%{matched_var_name}=
%{tx.0}"

SecRule &TX:ANOMALY_SCORE "@eq 0" "phase:5,t:none,log,pass,
logdata:'%{tx.bayes_msg}',exec:/etc/httpd/modsecurity.d/
bayes_train_ham.lua"
```

When we first deploy the rules, we run only the two training rules so that we may populate our corpus with real data from clients interactive with our unique web application. The rule listed last executes the bayes_train_ham.lua script when no malicious anomaly score is detected. Figure 1-5 shows a sample web application form for a loan application.

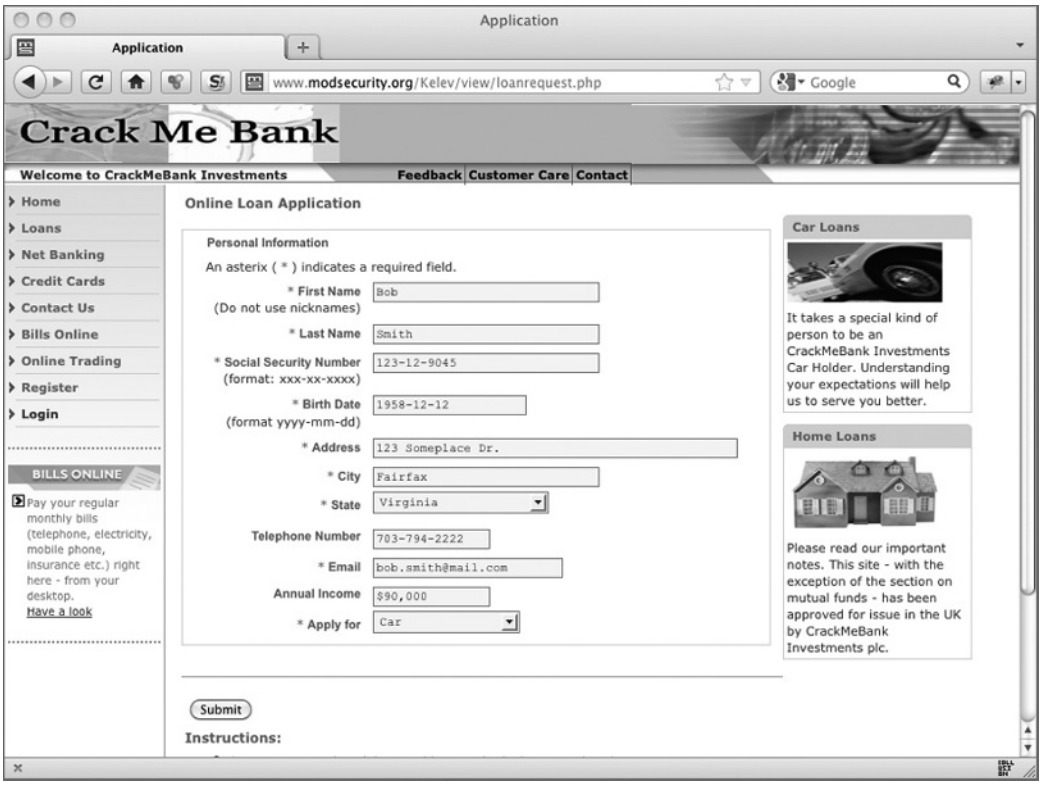

Figure 1-5: Sample loan application

When the client submits this form, the OWASP ModSecurity CRS attack signatures inspect each parameter value. If no malicious data is found, the bayes_train_ham.lua script trains the Bayesian ham classifier on each value:

```
Lua: Executing script: /etc/httpd/modsecurity.d/
bayes_train_ham.lua
 Arg Name: ARGS:txtFirstName and Arg Value: Bob.
 Arg Name: ARGS:txtLastName and Arg Value: Smith.
 Arg Name: ARGS:txtSocialScurityNo and Arg Value: 123-12-9045.
 Arg Name: ARGS:txtDOB and Arg Value: 1958-12-12.
 Arg Name: ARGS:txtAddress and Arg Value: 123 Someplace Dr..
 Arg Name: ARGS:txtCity and Arg Value: Fairfax.
 Arg Name: ARGS:drpState and Arg Value: VA.
 Arg Name: ARGS:txtTelephoneNo and Arg Value: 703-794-2222.
 Arg Name: ARGS:txtEmail and Arg Value: bob.smith@mail.com.
 Arg Name: ARGS:txtAnnualIncome and Arg Value: $90,000.
 Arg Name: ARGS:drpLoanType and Arg Value: Car.
 Arg Name: ARGS:sendbutton1 and Arg Value: Submit.
 Low Bayesian Score: . Training payloads as non-malicious.
```

```
 Setting variable: tx.bayes_msg=Training payload as ham: Submit.
 Set variable "tx.bayes_msg" to "Training payload as ham: Submit."
Lua: Script completed in 5647 usec, returning: Training payloads
as non-malicious: Submit..
Resolved macro %{tx.bayes_msg} to: Training payload as ham: Submit
Warning. Operator EQ matched 0 at TX. [file "/etc/httpd/
modsecurity.d/crs/base_rules/modsecurity_crs_48_bayes_analysis.
conf"
```

However, if an attacker inserts some malicious code into the Social Security Number field of that same form, the SQL Injection signatures of the ModSecurity CRS flag the payload, and the bayes_train_spam.lua script trains the classifier that this is spam. Here is a sample section from the modsec_debug.log file:

```
Lua: Executing script: /etc/httpd/modsecurity.d/
bayes_train_spam.lua
 Set variable "MATCHED_VARS:950901-WEB_ATTACK/SQL_INJECTION-ARGS:
txtSocialScurityNo" value "123-12-9045' or '2' < '5' ;--" size 29
to collection.
 Arg Name: MATCHED_VARS:950901-WEB_ATTACK/SQL_INJECTION-ARGS:
txtSocialScurityNo and Arg Value: 123-12-9045' or '2' < '5' ;--.
 Train Results: {misclassified=false,reinforced=true}.
 Setting variable: tx.bayes_msg=Completed Bayesian SPAM Training
on Payload: 123-12-9045' or '2' < '5' ;--.
 Set variable "tx.bayes_msg" to "Completed Bayesian SPAM Training
on Payload: 123-12-9045' or '2' < '5' ;--.".
 Lua: Script completed in 2571 usec, returning: Completed Bayesian
 SPAM Training on Payload: 123-12-9045' or '2' < '5' ;--..
 Resolved macro %{tx.bayes_msg} to: Completed Bayesian SPAM
Training on Payload: 123-12-9045' or '2' < '5' ;--.
 Warning. Pattern match ".*" at TX:950901-WEB_ATTACK/SQL_INJECTION
-ARGS:
txtSocialScurityNo. [file "/etc/httpd/modsecurity.d/crs/base_rules
/modsecurity_crs_48_bayes_analysis.conf"
```

Once you have let the Bayesian classifier training rules train on normal user traffic for a period of time, it is recommended that you run a web application scanning tool to help train the spam classifier for attack data. When this is done, you can activate the `SecRuleScript` rule that runs the bayes_check_spam.lua script. With this script activated, a request that did not trigger any previous rules has its payloads checked against the Bayesian classifier. The following is an example of an alert message that would be generated:

```
[Sun Feb 19 14:16:12 2012] [error] [client 72.192.214.223]
ModSecurity: Warning. Bayesian Analysis Alert for ARGS:
txtSocialScurityNo with payload: "345-22-0923'
 -10 union select 1,2,3,4,5,concat(user,char(58),password),7,8,9,
10 from mysql.user" [file "/etc/httpd/modsecurity.d/crs/base_rules
```

```
/modsecurity_crs_48_bayes_analysis.conf"] [line "3"] [msg
"Bayesian Analysis Detects Probable Attack."] [data "Score:
{prob=0.99968113864209,
probs={0.99968113864209,0.00031886135790698},class=\\x22/var/log/
httpd/spam
\\x22,pR=2.0627931680898,reinforce=true}"] [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 "www.modsecurity.org"]
 [uri "/Kelev/view/updateloanrequest.php"] [unique_id
"2V30csCo8AoAAHP5GBMAAAAB"]
```

[7] http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual

[8] https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project

[9] http://osbf-lua.luaforge.net/

[10] http://www.siefkes.net/software/moonfilter/

[11] http://www.siefkes.net/software/moonfilter/moonrunner.lua

[12] http://www.paulgraham.com/spam.html

[13] http://www.paulgraham.com/better.html

HTTP Audit Logging

RECIPE 1-6: ENABLE FULL HTTP AUDIT LOGGING

This recipe shows you how to capture full HTTP transaction data by using the ModSecurity audit engine.

Ingredients

- ModSecurity Reference Manual[14]
  - `SecRuleEngine` directive
  - `SecAuditEngine` directive
  - `SecAuditLog` directive
  - `SecAuditLogType` directive
  - `SecAuditLogParts` directive
  - `SecAuditLogStorageDir` directive
  - `SecRequestBodyAccess` directive
  - `SecResponseBodyAccess` directive
  - Audit log format documentation[15]

Enabling Full Audit Logging to One File

If you want to provide the greatest amount of data for incident response processes, you should enable full audit logging of both HTTP request and response traffic. Add or update the following ModSecurity directives in your Apache configuration files:

```
SecRuleEngine DetectionOnly
SecRequestBodyAccess On
SecResponseBodyAccess On
SecAuditEngine On
SecAuditLogParts ABCEFHZ
SecAuditLog /usr/local/apache/logs/audit.log
SecAuditLogType Serial
```

These directives create full audit logs of the HTTP transactions and store data from all clients to one file called /usr/local/apache/logs/audit.log. `SecAuditLogParts` defines the separate transactional elements that are captured:

- A: Audit log header
- B: Request headers
- C: Request body
- E: Intended response body
- F: Response headers
- H: Audit log trailer
- Z: Audit log footer

Let's again look at one of the WordPress `POST` web requests from before, except this time captured by ModSecurity's audit engine:

```
--26b60826-A--
[15/Feb/2012:09:08:17 --0500] Tzu8UcCoqAEAAR4rI1cAAAAA
109.70.36.102 58538 192.168.1.111 80
--357b3215-B--
POST /wordpress//xmlrpc.php HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: localhost
User-Agent: Wordpress Hash Grabber v2.0libwww-perl/6.02
Content-Length: 738

--357b3215-C--
<?xml version="1.0"?><methodCall><methodName>mt.setPostCategories
</methodName>   <params>   <param><value><string>3 union all
select user_pass from wp_users where id=3</string></value>
</param>   <param><value><string>admin</string></value>
</param>   <param><value><string>admin</string></value>
</param>   <param><value>  <array>    <data><value>  <struct>
<member>      <name>categoryId</name>      <value><string>1
```

```
</string></value>    </member>    <member>
<name>categoryName</name>      <value><string>Uncategorized
</string></value>    </member>    <member>      <name>isPrimary
</name>      <value><boolean>0</boolean></value>    </member>
</struct></value>  </data></array></value>   </param>   </params>
</methodCall>
--357b3215-F--
HTTP/1.1 200 OK
X-Powered-By: PHP/5.3.2-1ubuntu4.5
Content-Length: 649
Vary: Accept-Encoding
Content-Type: text/xml
Connection: close

--357b3215-E--
<div id='error'>
            <p class='wpdberror'><strong>WordPress database error
:</strong> [You have an error in your SQL syntax; check the manual
 that corresponds to your MySQL server version for the right
syntax to use near 'union all select user_pass from wp_users where
 id=3' at line 3]<br />
            <code>
                  DELETE FROM wp_post2cat
                  WHERE category_id = 349508cb0ff9325066aa6c490c
33d98b
                        AND post_id = 3 union all select user_
pass from wp_users where id=3
                  </code></p>
            </div><?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value>
        <boolean>1</boolean>
      </value>
    </param>
  </params>
</methodResponse>

--357b3215-H--
Apache-Handler: proxy-server
Stopwatch: 1329314896780667 97446 (- - -)
Stopwatch2: 1329314896780667 97446; combined=278, p1=9, p2=229,
p3=10, p4=11, p5=18, sr=0, sw=1, l=0, gc=0
Response-Body-Transformed: Dechunked
Producer: ModSecurity for Apache/2.7.0-dev1 (http://www.
modsecurity.org/).
Server: Apache/2.2.17 (Unix) mod_ssl/2.2.12 OpenSSL/0.9.8r DAV/2

--357b3215-Z--
```

As you can see, the ModSecurity audit log file captures the entire HTTP transaction. If you look at the bold section in Section C, Request Body, you can see that it looks like a SQL Injection attack. This code is attempting to manipulate the input in the hopes of altering the back-end SQL query logic to identify the local OS user whom the database is running as. In Section E, Response Body, we can see that the bold text shows that the database generated some error messages. We also see that the SQL Injection query executed and that the attacker could extract the password hash for the user. The attacker can now run various password-cracking sessions to try to enumerate the user's password. With this full transactional data captured, we are better equipped to figure out what data the attackers stole.

Enabling Full Audit Logging to Separate Files

Although it is convenient to have all transactional data logged to only one file with the `SecAuditLogType` Serial directive setting, there are two drawbacks. First, the file's size will grow extremely fast, depending on the amount of traffic your web application receives. You need to keep a close eye on this file size so that no problems with disk space or individual file size limits will occur. If the audit.log file exceeds this size limitation, new data is not appended to the log. The second potential issue is performance. As the name implies, with the Serial setting, each Apache child thread waits its turn to log to this file. For better performance, we should use the Concurrent setting:

```
SecRuleEngine DetectionOnly
SecRequestBodyAccess On
SecResponseBodyAccess On
SecAuditEngine On
SecAuditLogParts ABCIFEHZ
SecAuditLog /usr/local/apache/logs/audit.log
SecAuditLogType Concurrent
SecAuditLogStorageDir /usr/local/apache/audit/logs/audit
```

When running in Concurrent mode, each HTTP transaction is assigned its own audit log file. This approach provides better performance under heavy load and also facilitates central logging of data, which is described in Recipe 1-10. When running in Concurrent mode, the audit.log file's contents change from holding transactional data to instead working as an index file that points to the location of the individual files under the `SecAuditLogStorageDir` location:

```
localhost 127.0.0.1 - - [15/Feb/2012:14:35:41 --0500] "GET /wordpr
ess//xmlrpc.php HTTP/1.1" 200 %ld "-" "-" TzwJDcCoqAEAACQmHRAAAAAD
 "-" /20120215/20120215-1435/20120215-143541-TzwJDcCoqAEAACQmHRAAA
AAD 0 863 md5:ea8618293f59d2854d868685445cd4c8
localhost 127.0.0.1 - - [15/Feb/2012:14:35:42 --0500] "POST /wordp
ress//xmlrpc.php HTTP/1.1" 200 %ld "-" "-" TzwJDsCoqAEAACQjHTIAAAA
A "-" /20120215/20120215-1435/20120215-143542-TzwJDsCoqAEAACQjHTIA
AAAA 0 2220 md5:ed4231c6d2f1af4a1bf4cef11481f28f
```

Each transaction uses the Apache `mod_uniqueid` hash in its filename to allow for identification. Each file still holds the exact same data as in Serial mode, except that it holds data from only one transaction.

[14] `http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual`

[15] `https://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Data_Format`

RECIPE 1-7: LOGGING ONLY RELEVANT TRANSACTIONS

This recipe shows you how to configure ModSecurity to log only transactions that are deemed relevant from a security perspective.

Ingredients

- ModSecurity Reference Manual[16]
  - `SecRuleEngine` directive
  - `SecAuditEngine` directive
  - `SecAuditLog` directive
  - `SecAuditLogType` directive
  - `SecAuditLogParts` directive
  - `SecAuditLogStorageDir` directive
  - `SecRequestBodyAccess` directive
  - `SecResponseBodyAccess` directive
  - `SecAuditLogRelevantStatus` directive

I strongly recommend that organizations use full HTTP audit logging, as described in Recipe 1-6. That being said, I understand that logging full HTTP transactional data may be infeasible for your web application. If you decide not to log all data, you can configure ModSecurity to log only what it determines to be *relevant* transactions. If you change the `SecAuditEngine` directive from `On` to `RelevantOnly`, ModSecurity creates an audit log entry under only two distinct scenarios:

- If there is a positive match from one of the `SecRule` directives
- If the web server responds with an HTTP status code as defined by a regular expression in the `SecAuditLogRelevantStatus` directive

Here is an updated audit logging configuration that uses only relevant logging:

```
SecRuleEngine DetectionOnly
SecRequestBodyAccess On
SecResponseBodyAccess On
SecAuditEngine RelevantOnly
```

```
SecAuditLogRelevantStatus "^(?:5|4(?!04))"
SecAuditLogParts ABCIFEHZ
SecAuditLog /usr/local/apache/logs/audit.log
SecAuditLogType Serial
```

With these configurations, in addition to normal ModSecurity `SecRule` matches, audit logs are created for any transaction in which the HTTP response status code is a 500 level (server errors) or 400 level (user errors), excluding 404 Not Found events.

[16] `http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual`

RECIPE 1-8: IGNORING REQUESTS FOR STATIC CONTENT

This recipe shows you how to configure ModSecurity to exclude audit logging of HTTP requests for static resources.

Ingredients

- ModSecurity Reference Manual[17]
  - `ctl:ruleEngine` action
  - `ctl:auditEngine` action

Logging all HTTP transactions is ideal from an incident response perspective. However, some organizations may decide that they want to exclude inspection and logging of requests for static resources to improve performance and latency and reduce the amount of logging required. The theory is that if a request for some type of static resource (such as image files) occurs, the potential attack surface is greatly reduced, because there are no parameters. Parameter payloads are used as the primary injection points for passing attack data to dynamic resources that accept user input for internal processing. If we want to disable inspection and logging for these static resource requests, we must first analyze the request components to ensure that they are not attempting to pass any parameter data. Take a look at the following sample rules:

```
SecRule REQUEST_METHOD "@pm GET HEAD" "id:'999001',chain,phase:1,
t:none,nolog,pass"
        SecRule REQUEST_URI "!@contains ?" "chain"
                SecRule &ARGS "@eq 0" "chain"
                        SecRule &REQUEST_HEADERS:Content-Length|
&REQUEST_HEADERS:Content-Type "@eq 0" "ctl:ruleEngine=Off,
ctl:auditEngine=Off"
```

This chained rule set verifies the request details by doing the following:

- It verifies that the request method is either a `GET` or a `HEAD`. If it is anything else, the request should probably be logged, because it is a dynamic request method looking to alter data.
- It verifies that no query string is present in the URI by checking for the question mark character.
- It verifies that no parameters are present in the query string or request body by checking for the presence of the `ARGS` collection.
- It verifies that no request body is present by checking for the existence of the `Content-Length` and `Content-Type` request headers.

If all these rules match, the final `SecRule` executes the two bold `ctl` actions to dynamically disable the rule and audit engines.

CAUTION

The rationale for disabling inspection and logging of static resource requests is valid, but you should approach this choice with caution. Although these rules help profile the potential attack surface, they are not foolproof. The main attack vector location, which is still open, is *cookies*. If your application uses cookies, this leaves open a potential vector for attacks. However, if you update the sample exclusion rules to include checking for the existence of the `Cookie:` request header, you lose the performance gain you are going for, because cookies are sent for static image requests.

[17] `http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual`

RECIPE 1-9: OBSCURING SENSITIVE DATA IN LOGS

This recipe shows you how to use ModSecurity to obscure sensitive data that is captured within the audit logs.

Ingredients

- ModSecurity Reference Manual[18]
  - `sanitiseArg` action
  - `sanitiseMatchedBytes` action

HTTP audit logging comes with a catch-22: Some sensitive user data probably is captured within the logs, such as passwords or credit card data. We want to log these transactions, but we do not want to expose this sensitive data to anyone who may access these logs. ModSecurity has a few different rule actions that can be used to obscure selected data within the audit logs. Let's look at a few sample use cases.

Login Passwords

Let's use the WordPress login form shown in Figure 1-6 as an example.

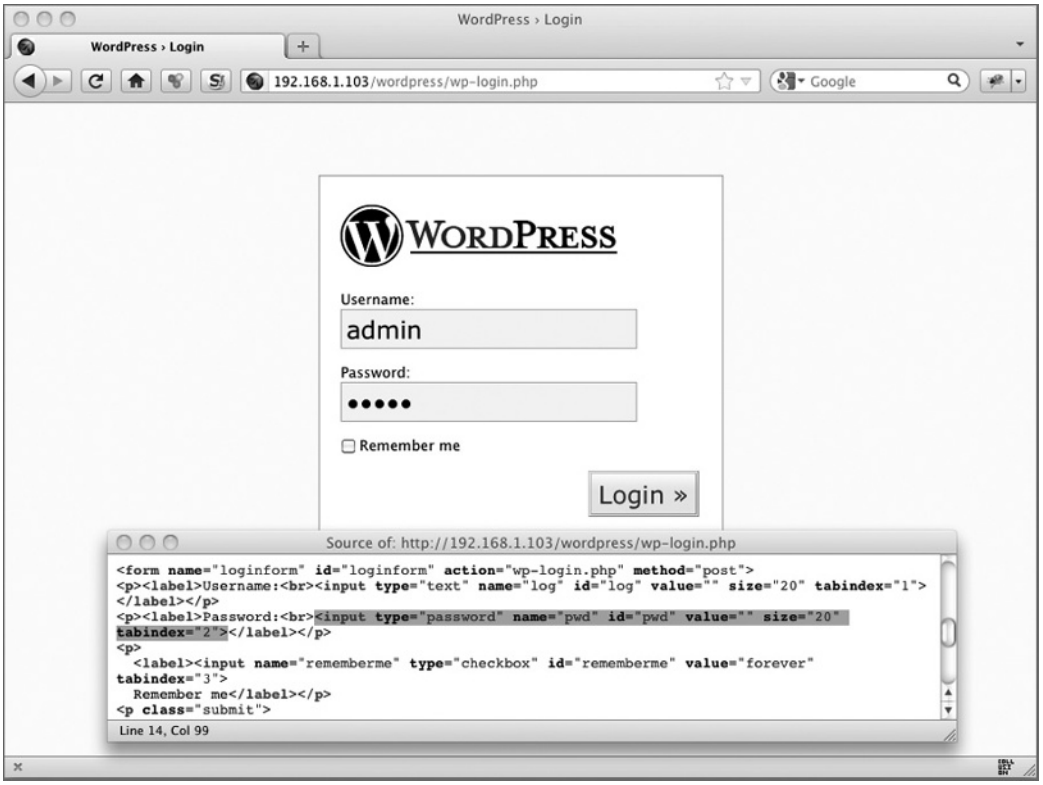

Figure 1-6: Sample WordPress login form

In Figure 1-6, we see from the HTML source code that the password being submitted is passed in a parameter called `pwd`. If we want to obscure the password payload in the logs, we can add the following sample rule that uses the ModSecurity `sanitiseArg` action:

```
SecRule &ARGS:pwd "@eq 1" "phase:5,t:none,id:'111',nolog,pass,\
sanitiseArg:pwd"
```

When the login form is submitted, this is how the transactional data now appears in the audit log file:

```
--e947184b-A--
[20/Feb/2012:09:54:24 --0500] T0Jen8CoAWcAAQ0jNzcAAAAA 192.168.1.103
 59884 192.168.1.103 80
--e947184b-B--
POST /wordpress/wp-login.php HTTP/1.1
Host: 192.168.1.103
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:10.0.1)
Gecko/20100101 Firefox/10.0.1
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
DNT: 1
Connection: keep-alive
Referer: http://192.168.1.103/wordpress/wp-login.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 63

--e947184b-C--
log=admin&pwd=*****&submit=Login+%C2%BB&redirect_to=wp-admin%2F
--e947184b-F--
HTTP/1.1 302 Found
X-Powered-By: PHP/5.3.2-1ubuntu4.5
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Last-Modified: Sat, 18 Feb 2012 09:32:29 GMT
Cache-Control: no-cache, must-revalidate, max-age=0
Pragma: no-cache
Location: wp-admin/
Vary: Accept-Encoding
Content-Length: 0
Content-Type: text/html; charset=UTF-8
Set-Cookie: wordpressuser_fdd6fe9e4093f5711cf9621dd3ae90d9=admin;
path=/wordpress/
Set-Cookie: wordpresspass_fdd6fe9e4093f5711cf9621dd3ae90d9=
c3284d0f94606de1fd2
af172aba15bf3; path=/wordpress/
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
```

As you can see from the bold data in Section C, the `pwd` parameter payload is now obscured with asterisks.

Credit Card Usage

If your web application conducts e-commerce transactions in which users submit credit card data, you need to be sure to also obscure that data within the audit logs. The following ModSecurity rules sanitize any payload that passes the `@verifyCC` operator check:

```
SecRule ARGS "@verifyCC \d{13,16}" "id:'112',phase:2,log,capture,\
pass,msg:'Credit Card Number Detected in Request',sanitiseMatched"
```

This example uses `sanitiseMatched` instead of the specific `sanitiseArg` action because organizations often are not completely sure of every possible parameter location within their applications where credit card data may be submitted. Here is a sample audit log entry showing that the bold `creditCardNumber` parameter payload is now obscured:

```
--4358a809-A--
[20/Feb/2012:11:02:53 --0500] T0Jup8CoqAEAAUFmEEgAAAAA 127.0.0.1
60880 127.0.0.1 80
--4358a809-B--
POST /site/checkout.jsp HTTP/1.1
Host: www-ssl.site.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:10.0.1)
Gecko/20100101 Firefox/10.0.1
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
DNT: 1
Connection: keep-alive
Referer: https://www-ssl.site.com/site/cart.jsp
Cookie: JSESSIONID=CD052245017816ABD24D4FD2C836FAD9;
Content-Type: application/x-www-form-urlencoded
Content-Length: 6020

--4358a809-C--
paymentType=default&_D%3ApaymentType=+&
creditCardNumber=**********&_D%3AcreditCardNumber=+&cid=6802
&_D%3Acid=+&selCreditCardType=AmericanExpress&_D%3AselCreditCardType
=+&_D%3AexpirationMonth=+&expirationMonth=10&_D%3
AexpirationYear=+&expirationYear=2013&_D%3
AchkSaveCreditCard=+&financeOptions=14&_D%3AfinanceOptions=+&_D%3
AfinanceOptions=+&_D%3AfinanceOptions=+&...
```

[18] http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual

Centralized Logging

RECIPE 1-10: SENDING ALERTS TO A CENTRAL LOG HOST USING SYSLOG

This recipe shows you how to configure Apache to use Syslog to send alert data to a central logging host.

Ingredients

- Apache `ErrorLog` directive[19]
- Syslog configuration file
- Central Syslog host

The standard logging mechanism for ModSecurity uses the local file system for storage. The short, one-line alert messages are automatically logged to the Apache `ErrorLog` directive location. This approach works fine for smaller organizations, but if you have an entire web server farm to monitor, it can quickly become unmanageable to keep track of alerts. In this scenario, you can quite easily reconfigure your Apache settings to send its error_log data to the local Syslogd process for handling. This is quickly accomplished by updating the `ErrorLog` directive like this:

```
$ grep ErrorLog httpd.conf
#ErrorLog /var/log/httpd/error_log
ErrorLog syslog:local7
```

This new setting sends all Apache error messages to the local syslog process using the `local7` facility. The next step is to edit the syslog.conf file and add some new entries:

```
$ grep local7 /etc/syslog.conf
local7.*                    /data/httpd/error_log
local7.*                    @192.168.1.200
```

Here we have added two entries to the syslog.conf file. The first entry simply reroutes the Apache error_log data to the normal local file location on the host. The second entry forwards all the same data to the central logging host at IP address 192.168.1.200 using the default UDP protocol on port 514. After these settings are added, you should restart your Syslogd service. After this is done, the Apache error_log data should be sent to the central logging host. Here is some sample output after the ngrep tool on the ModSecurity sensor host has been used to monitor the data as it is sent to the central log host using Syslog:

```
$ sudo ngrep -d eth5 port 514
interface: eth5 (192.168.1.0/255.255.255.0)
```

```
filter: (ip or ip6) and ( port 514 )
#
U 192.168.1.110:514 -> 192.168.1.200:514
  <187>httpd[16219]: [error] [client 192.168.1.103] ModSecurity:
Warning. Pattern match "^[\\\\d.:]+$" at REQUEST_HEADERS:Host.
[file "/opt/wasc-honeypot/etc/crs/activated_rules/
modsecurity_crs_21_protocol_anomalies.conf"] [line
"98"] [id "960017"] [rev "2.2.1"] [msg "Host header is a numeric
IP address"] [severity "CRITICAL"] [tag "PROTOCOL_VIOLATION/IP_
HOST"] [tag "WASCTC/WASC-21"] [tag "OWASP_TOP_10/A7"]
[tag "PCI/6.5.10"] [tag "http://technet.microsoft.com/en-
us/magazine/2005.01.hackerbasher.aspx"]
[hostname "192.168.1.110"] [uri "/wp-content/themes
/pbv_multi/scripts/timthumb.php"] [unique_id
"T0J-M38AAQEAAD9bDpAAAAAA"]
#
U 192.168.1.110:514 -> 192.168.1.200:514
  <187>httpd[16219]: [error] [client 192.168.1.103] ModSecurity:
Warning. Match of "beginsWith %{request_headers.host}" against
"TX:1" required. [file "/opt/wasc-honeypot
/etc/crs/activated_rules/modsecurity_crs_40_generic_attacks.
conf"] [line "168"] [id "950120"] [rev "2.2.1"] [msg "Remote
File Inclusion Attack"] [severity "CRITICAL"] [hostname
"192.168.1.110"] [uri "/wp-content/themes/pbv_multi/scripts/
timthumb.php"] [unique_id "T0J-M38AAQEAAD9bDpAAAAAA"]
#
U 192.168.1.110:514 -> 192.168.1.200:514
  <187>httpd[16219]: [error] [client 192.168.1.103] ModSecurity:
Warning. Pattern match "\\\\bsrc\\\\b\\\\W*?\\\\bhttp:" at
REQUEST_URI.
 [file "/opt/wasc-honeypot/etc/crs/activated_rules/
modsecurity_crs_41_xss_attacks.conf"] [line "405"] [id "958098"]
[rev "2.2.1"] [msg "Cross-site Scripting (XSS) Attack"] [data
"src=http:"] [severity "CRITICAL"] [tag "WEB_ATTACK/XSS"]
[tag "WASCTC/WASC-8"] [tag "WASCTC/WASC-22"]
[tag "OWASP_TOP_10/A2"] [tag "OWASP_AppSensor/IE1"]
[tag "PCI/6.5.1"] [hostname "192.168.1.110"]
[uri "/wp-content/themes/pbv_multi/scripts/timthumb.php"]
 [unique_id "T0J-M38AAQEAAD9bDpAAAAAA"]
```

Figure 1-7 shows the Syslog data in a central logging host using the Trustwave Security Information Event Management (SIEM)[20] application.

After these ModSecurity alerts are centralized, custom searching and alerting mechanisms may be implemented to conduct further analysis and trending information for events from across your web architecture.

[19] http://httpd.apache.org/docs/2.2/mod/core.html#errorlog

[20] https://www.trustwave.com/siem/

Recipe 1-10

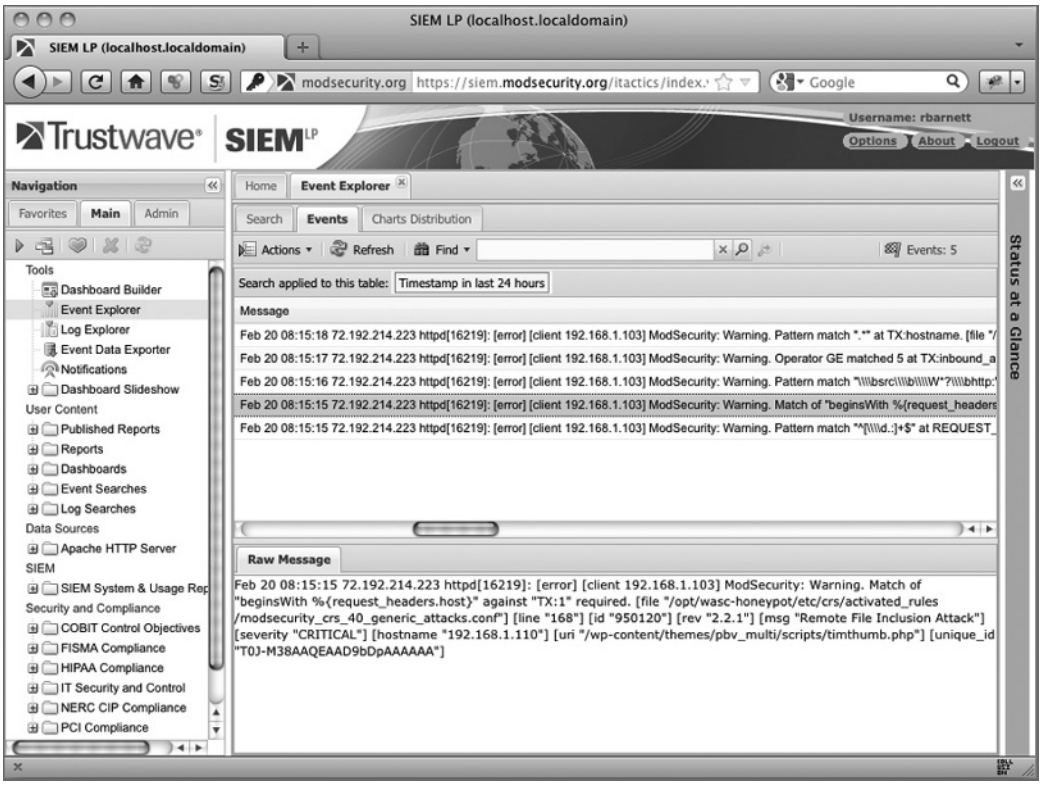

Figure 1-7: Syslog data in Trustwave's SIEM

Recipe 1-11

RECIPE 1-11: USING THE MODSECURITY AUDITCONSOLE

This recipe shows you how to set up the ModSecurity AuditConsole for centralized logging of audit log data.

Ingredients

- Jwall AuditConsole[21]
- ModSecurity's mlogc program

Recipe 1-10 showed you how to centralize the short, one-line ModSecurity alert message that is sent to the Apache error_log file by sending it through Syslog. This is a good

approach, but the main disadvantage is that the data being centrally logged is only a small subset of the data that was logged in the audit log file. To confirm the accuracy of the alert messages, you need to review the full audit log file data. One application that can be used for central logging of ModSecurity events is AuditConsole, a Java tool written by Christian Bockermann.

Installation

Here ar e the basic steps for installing the AuditConsole. First, download the latest version of the console from `http://download.jwall.org/AuditConsole/current/`. Next, you need to choose a location where you want the console to be installed. The following commands assume that you will place it under the /opt directory:

```
# cd /opt
# unzip /path/to/AuditConsole-0.4.3-16-standalone.zip
# cd /opt/AuditConsole
# chmod 755 bin/*.sh
```

At the time this book was written, the latest version was 0.4.3-16.

The `chmod` command is required, because zip archives normally do not preserve the executable bit required on the scripts under the bin/ directory. The final step is to start the console, log into the web interface, and run through the setup wizard:

```
# cd /opt/AuditConsole
# sh bin/catalina.sh start
```

After the application starts, use a web browser to go to `https://localhost:8443` and follow the setup wizard instructions. The default back-end database is Apache Derby. It is recommended that you change the database to MySQL for production usage. It is also recommended that you follow the excellent User Guide documentation on the Jwall web site to configure all aspects of the AuditConsole: `https://secure.jwall.org/AuditConsole/user-guide/`.

To set up a remote sensor, go to the System ➢ Sensors location, click the Add Sensor button, and fill in the data, as shown in Figure 1-8.

As soon as you have added a remote sensor to the AuditConsole, the next step is to update the ModSecurity host's audit log configurations so that it can forward its logs. Update the audit log settings as shown here so that the `SecAuditLog` directive points to the mlogc program and the mlogc.conf file:

```
SecRuleEngine DetectionOnly
SecRequestBodyAccess On
SecResponseBodyAccess On
SecAuditEngine On
SecAuditLogParts ABCIFEHZ
```

```
SecAuditLog "|/user/local/bin/mlogc /usr/local/apache
/etc/mlogc.conf"
SecAuditLogType Concurrent
SecAuditLogStorageDir /usr/local/apache/audit/logs/audit
```

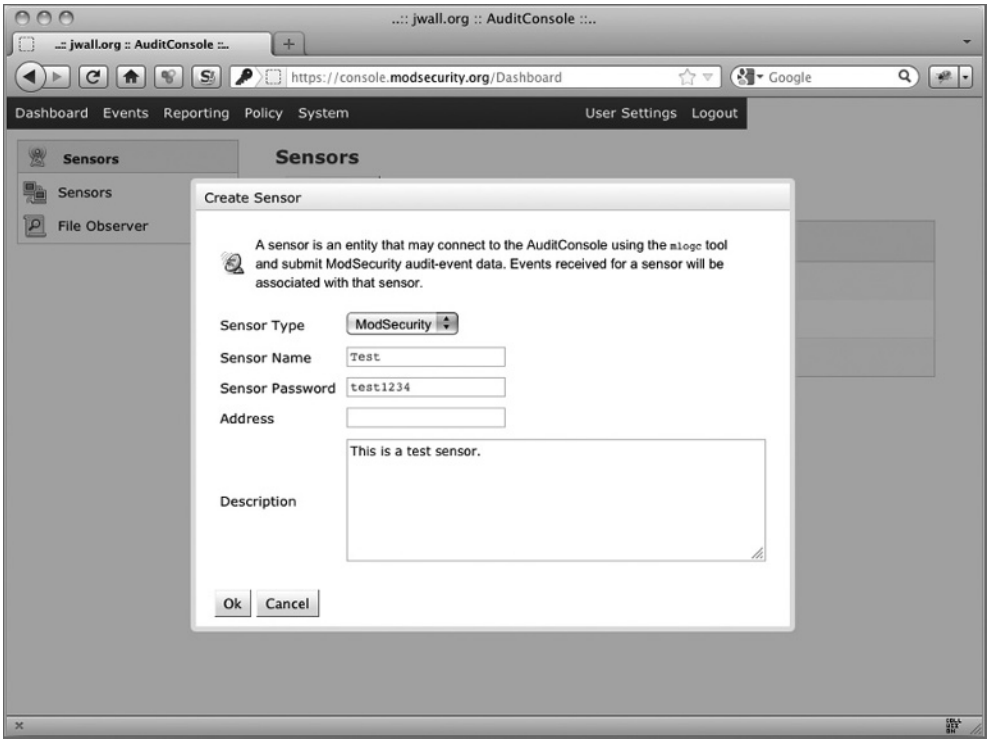

Figure 1-8: AuditConsole's Add Sensor page

The mlogc program acts as an HTTP client utility. As new audit log data is created, it forwards the individual audit log files to the central logging host by using an HTTP/HTTPS `PUT` method and then uploads the file. The mlogc.conf file is the configuration file where you can specify how to manage the audit logs. The following is an example of the mlogc.conf configuration file. The bold entries are the most relevant. `ConsoleURI` points to the location of the central AuditConsole host, and `SensorUsername` and `SensorPassword` are the credentials you specified when creating the sensor shown in Figure 1-8:

```
##########################################################################
# Required configuration
#   At a minimum, the items in this section will need to be adjusted
#   to fit your environment.  The remaining options are optional.
##########################################################################
```

```
# Points to the root of the installation. All relative
# paths will be resolved with the help of this path.
CollectorRoot       "/data/mlogc"

# ModSecurity Console receiving URI. You can change the host
# and the port parts but leave everything else as is.
ConsoleURI          "http://192.168.1.201/rpc/auditLogReceiver"

# Sensor credentials
SensorUsername      "Test"
SensorPassword      "test1234"

# Base directory where the audit logs are stored.  This can be
# specified as a path relative to the CollectorRoot, or a full path.
LogStorageDir       "data"

# Transaction log will contain the information on all log collector
# activities that happen between checkpoints. The transaction log
# is used to recover data in case of a crash (or if Apache kills
# the process).
TransactionLog      "mlogc-transaction.log"

# The file where the pending audit log entry data is kept. This file
# is updated on every checkpoint.
QueuePath           "mlogc-queue.log"

# The location of the error log.
ErrorLog            "mlogc-error.log"

# Keep audit log entries after sending? (0=false 1=true)
# NOTE: This is required to be set in SecAuditLog mlogc config if
# you are going to use a secondary console via SecAuditLog2.
KeepEntries         0


######################################################################
# Optional configuration
######################################################################

# The error log level controls how much detail there
# will be in the error log. The levels are as follows:
#   0 - NONE
#   1 - ERROR
#   2 - WARNING
#   3 - NOTICE
#   4 - DEBUG
#   5 - DEBUG2
```

```
#
ErrorLogLevel       3

# How many concurrent connections to the server
# are we allowed to open at the same time? Log collector uses
# multiple connections in order to speed up audit log transfer.
# This is especially needed when the communication takes place
# over a slow link (e.g. not over a LAN).
MaxConnections      10

# The time each connection will sit idle before being reused,
# in milliseconds. Increase if you don't want ModSecurity Console
# to be hit with too many log collector requests.
TransactionDelay    50

# The time to wait before initialization on startup in milliseconds.
# Increase if mlogc is starting faster than termination when the
# sensor is reloaded.
StartupDelay    1000

# How often is the pending audit log entry data going to be written
# to a file? The default is 15 seconds.
CheckpointInterval  15

# If the server fails all threads will back down until the
# problem is sorted. The management thread will periodically
# launch a thread to test the server. The default is to test
# once in 60 seconds.
ServerErrorTimeout  60

# The following two parameters are not used yet, but
# reserved for future expansion.
# KeepAlive         150
# KeepAliveTimeout  300
```

When everything is configured and you have restarted Apache, new audit log files are sent to the AuditConsole host in real time. You can then use the AuditConsole to view, sort, and search for events of interest and see full audit log details, as shown in Figure 1-9.

[21] http://jwall.org/web/audit/console/index.jsp

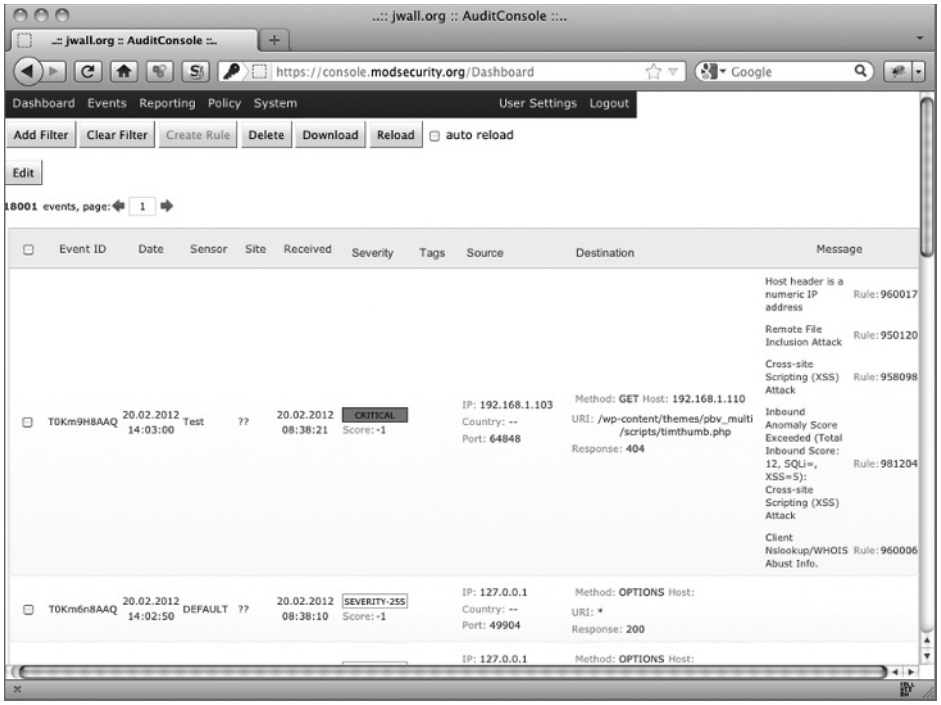

Figure 1-9: AuditConsole's Events page