Tactical Defense
with ModSecurity

– Exercises –

August 2013, Hamburg

Christian Bockermann
chris@jwall.org

Contents

0. Preparations - Basic Startup

Exercise 0.1: Prepare virtual Machine

To get a grip, we first need to create a virtual machine image within VirtualBox. All required software and images are provided on the Workshop USB Stick.

1. **Install & Start VirtualBox**

2. **Create a Virtual Machine**
   The Workshop USB stick contains an `ModSecurity.ova` file, which is an appliance image for VirtualBox.

   a) (*Optional:* Copy the `ModSecurity.ova` file onto your hard drive.)

   b) Start *VirtualBox* and open the settings. Under *network settings* add a new *Host-only network* where we can connect the ModSecurity machine.

   c) In the *VirtualBox* menu *File* choose *Import Appliance* to create a new virtual machine instance based on the `ModSecurity.ova` file from the USB stick.

      Important: Please disable *USB 2.0* support when importing the appliance.

3. **Start the Virtual Machine**
   You should be able to start the virtual machine inside VirtualBox now and log into it as

   user `modsecurity` with password `modsecurity`.

Your Notes:

Exercise 0.2: Browser Access

The virtual machine contains a simple Java web-application which is running within a Tomcat 7 Servlet container. The container is started upon system-boot.
Since this will be the basis of all of the following exercises, make yourself comfortable with the booted system:

1. Determine the IP address of your virtual ModSecurity machine, e.g. by using the `ifconfig` command.

2. Access the simple Java web-app on the acquired IP address at port 8080 using your browser.

You should be able to access the system via *ssh*.

Your Notes:

Exercise 0.3: Setting up a Reverse-Proxy

The task of this exercise is to configure your Apache web-server to serve as a reverse-proxy in front of the sample Tomcat application server:

1. Check the configuration below `/etc/apache2`.
2. Enable the required proxy modules.
3. Extend the configuration of the default virtual host to serve the contents of the Tomcat 67 application server.
4. (Re-) Start Apache and test your configuration by accessing the server with a regular browser on port 80.

By the end you should be able to access the sample application using port 80. You may want to verify this by checking you Apache log-files.

Your Notes:

Exercise 0.4: Proxy-Pass Settings

Please recall again, that the `ProxyRequests` directive is *not* required for running Apache in reverse proxy mode as it only affects the *forwarding-proxy* capability of Apache.

1. Explicitly disable `ProxyRequests` and ensure that your Apache is still working as expected.
2. Switch the `ProxyPreserveHost` setting `On` and `Off` and check the effect in the demo application.
3. Try different settings for `ProxyVia`: `Off`, `On`, `Full` and check the demo application for the effects.

Your Notes:

1. ModSecurity Installation

Exercise 1.1: Prepare Directory Layout

Before compiling and installing ModSecurity, we will need to prepare a solid directory layout. Putting everything in the right place from the beginning will help to not loose ourselves.

1. Log into the Virtual Machine as user `modsecurity`.
2. Examine the file `samples/init-directories.sh` in the home directory and adjust the `USER_APACHE` and `GROUP_APACHE` variables to match the Apache user.
3. Execute this script using `sudo`.

You should verify the created directories in `/opt/modsecurity`.

Your Notes:

Exercise 1.2: Compile & Install ModSecurity

In this exercise we focus on the installation of ModSecurity from source. The source archive can be downloaded from `modsecurity.org`. The virtual machine contains the ModSecurity source of version 2.7.5 in the home directory of user `modsecurity`. Compiling ModSecurity is rather simple. You'll need to walk through the following steps to set up ModSecurity:

1. Log in as user `modsecurity`
2. Unpack the source-archive in this users' home directory
3. Run the `./configure` script to configure the sources with the `--prefix=/opt/modsecurity` option appended
4. Run `make` to compile the ModSecurity module
5. Run `sudo make install` to install the module

Your Notes:

Exercise 1.3: Compile & Install `mlogc`

The `mlogc` tool is the standard way to send audit-log data to a remote log console, such as the *AuditConsole*.

1. Locate the `mlogc` directory in your ModSecurity source directory and run

   ```
   # make
   ```

   to compile the `mlogc` tool.

2. Copy the `mlogc` tool to `/opt/modsecurity/bin/mlogc` by running

   ```
   # make install
   ```

Your Notes:

Exercise 1.4: Install the jwall-tools (optional)

The *jwall-tools* is a collection of simple commands written in Java which can be very convenient to manage ModSecurity audit-logs and configurations.

1. Check that the file `/etc/apt/sources.list` contains a line for the jwall.org Debian repository:

   ```
   deb http://download.jwall.org/debian jwall main
   ```

2. Download the GPG key of the repository and add it to the APT keyring:

   ```
   # wget http://download.jwall.org/chris.gpg
   # sudo apt-key add chris.gpg
   ```

3. Install the *jwall-tools* by running

   ```
   # sudo apt-get install jwall-tools
   ```

4. Test the proper installation by running

   ```
   # jwall -version
   ```

Your Notes:

Exercise 1.5: Create Minimal ModSecurity Configuration `main.conf`

Following the *ModSecurity Handbook* we will create a single *root*-configuration file called `main.conf`, which will be linked into in the Apache config. This file will then reference all ModSecurity related configurations.
With that concept it is easy to completely disable ModSecurity by un-linking that file.

1. Within the `modsecurity` user's home, check for the file `samples/main.conf` and copy that to `/opt/modsecurity/etc/main.conf`

   ```
   # cd /home/modsecurity/samples
   # cp main.conf /opt/modsecurity/etc/main.conf
   ```

Your Notes:

Exercise 1.6: Enable ModSecurity in Apache

Now, that we have the module in place and created a minimum configuration, we are ready to load the module into Apache and bring it alive.

1. Create `security2.load` in the `mods-available` directory of your Apache. That file needs to load the required libraries and the module itself:

   ```
   LoadFile /usr/lib/x86_64-linux-gnu/libxml2.so.2
   LoadFile /usr/lib/x86_64-linux-gnu/liblua5.1.so
   LoadModule security2_module \
   /opt/modsecurity/lib/mod_security2.so
   ```

   Create a link to this file inside `mods-enabled`.

2. Create the `security2.conf` which should simply include your main ModSecurity configuration file:

   ```
   Include /opt/modsecurity/etc/main.conf
   ```

   Create a link to this file inside `mods-enabled`

3. Check for a proper setup if your Apache configuration by running

   ```
   # apache2ctl configtest
   ```

Hint: *After you* **created** *the files, the links can easily be created using the* ***`a2enmod`*** *command.*
It is always a good idea to check the proper format of the Apache configurations before restarting Apache. The ***`apache2ctl configtest`*** *command does not catch any logical errors, but might save downtime due to mispelled commands or incorrectly loaded modules.*

Your Notes:

2. ModSecurity Setup

Exercise 2.1: Knowing what to log

The worst mistake in logging is to not log anything at all. To be selective, setup the *AuditEngine* of ModSecurity to suit your needs:

1. Enable full transaction logging for all requests. Access the Demo application and check the resulting `audit.log` file.
2. Turn off `SecRequestBodyAccess` and verify that the request bodies are no longer contained in the audit-log.
3. Set up ModSecurity to log the full response body and check the `audit.log`
4. Now, limit the transaction logging to non-20x responses only.

Your Notes:

Exercise 2.2: Logging Performance Impact

The *jwall-tools* allow for a re-injection of transactions read from a ModSecurity audit-log file. In this excercise we are interested in seeing the performance impact on various log settings:

1. Create a simple test-sequence of requests by running

   ```
   # cat /opt/modsecurity/var/audit/audit.log >> /tmp/test.log
   ```

   several times.

   Use the `count` command of the *jwall-tools* to check the number of events in your `test.log` file:

   ```
   # jwall count /tmp/test.log
   ```

2. Re-send the transactions in `test.log` using the *jwall-tools*' `eval`-command:

   ```
   # jwall eval -d 127.0.0.1 /tmp/test.log
   ```

   Run this several times and write down the request rates.

3. Change the `SecAuditLogParts` and the *relevant only* setting of the *AuditEngine* and re-run the evaluation (multiple times). Compare the request rates.

Your Notes:

Exercise 2.3: Request Body Processing

As mentioned in the presentation, ModSecurity needs to buffer the request body in order to inspect it during request processing. This requires memory and may result in denial of service attacks when flooding the server with large request.

1. Set the `SecRequestBodyNoFilesLimit` to a small value and use the demo application to submit a request body with the form. Try to exceed the request body limit value and check the response of ModSecurity.
2. Recall the `SecResponseBodyLimit` setting and modify this to a small value. Access the demo application and check the results in your audit log file.

Your Notes:

Exercise 2.4: The Scope of Settings

So far we've put all settings into the main Apache server scope. However, there are usually different requirements for different sites served within a single Apache instance. In this exercise we will set up a virtual host on port 81 with more production like settings.

1. Change into the directory `/etc/apache2/sites-available`
2. Copy the file `000-default` to `001-default` and define the virtual host in that new file to listen on port 81. Add the directive `Listen 81` to `/etc/apache2/ports.conf`.
3. Create a symbolic link to that file in `/etc/apache2/sites-enabled`.
4. Define the log-files for that host to be `default_81-access.log`, `default_81-audit.log` and `default_81-error.log` for the access-log, the ModSecurity audit-log and the Error-log respectively.
5. Restart Apache and access the demo application via port 81. Verify that the transactions are logged into the new log-files.

Your Notes:

3. ModSecurity Rule Language

Exercise 3.1: A simple ModSecurity Rule

So far we have set up an Apache web-server that includes the ModSecurity module. The module itself won't take any action on incoming requests until there are rules employed.

1. Create a simple rule that will check all parameters of requests for the string `DROP TABLE`. The rule should block/deny these requests.

2. Request the sample web-application and transmit data containing the string `DROP TABLE`.

3. Add a `msg` and an `id` to your rule. Test it again and check the logs for your messages.

Your Notes:

Exercise 3.2: Selectively raising the DebugLogLevel

It may sometimes be hard to find the correct messages in the logs when logging is too verbose. Here we create a simple rule which enables verbose logging for a specific URI only.
Use the following rule for verbose debugging of `/index.jsp` only:

```
# the global debug-log level
#
SecDebugLogLevel 0

# verbose debugging for /index.jsp
#
SecRule REQUEST_URI "@eq /index.jsp"   phase:1,pass,ctl:debugLogLevel=9
```

Your Notes:

Exercise 3.3: Blocking with Response Status

The actions `drop`, `deny` and `status` are important to block requests when a rule matches.

1. Alter your first rule to use the `deny` action to block a request.
2. Add an additional `status:XX` action to block the request with different response codes (replace `XX` with the response code, e.g. 411).
3. Replace `deny` and `status` with the `drop` action (you need to add the action `phase:2` for this.)

Your Notes:

Exercise 3.4: Evading our own Rule

Now we instantly want to evade our own rule. SQL injections do not care about case sensitivity.

1. Change your request parameter to `DRoP taBLE` or similar and try again. Is you rule still blocking?
2. Have a look at the debug-log!
3. Add the `t:lowercase` transformation to your rule and change your rule accordingly to block requests with any case-variants of `DROP TABLE`
4. What about adding spacing in between `DROP TABLE`?
   Check the list of transformations for a useful one to catch that evasion!
5. Thinking about newlines! Can be easily injected using `%0a%0d` in the URL. Use the `/objects.jsp` page of the demo application and use the text area to inject `DROP TABLE` split by newlines.

   Have a look at the debug-log and review how the newlines are handled!

Your Notes:

Exercise 3.5: Taking a different view

In the last exercise we used the `ARGS` collection to check parameters for attacks. To help you understand what that relies on, try to block the same attacks using the `QUERY_STRING` variable.

1. Use your browser to access the following URL on your virtual machine:

   ```
   /index.jsp?variable=id&value=DROP%0a%0dTABLE
   ```

   Now create a rule checking the `QUERY_STRING` for the `DROP TABLE` command.

2. Check the debug-log to see what data is used for matching the request.

3. Check the list of transformations for one that helps catching `DROP%0a%0dTABLE` using the `QUERY_STRING` variable.

Your Notes:

Exercise 3.6: User-Friendly Blocking

Simply blocking user-requests is a rather rude way of implying a security policy. In this exercise we want to extend the previous solution by showing blocked users a informative error-page.

1. Create a simple error page in `/var/www/_error_documents_/` and define this as an ErrorDocument within Apache, e.g:

   ```
   #
   # Error Page
   #
   ErrorDocument 509 /_error_documents_/blocked-509.html
   ```

2. Use the `deny` action in combination with the `status` action in your rules to block requests with the `DROP TABLE` command

3. Test your error-page by accessing the demo application accordingly.

Hint: *There already exists a prepared `_error_documents_` folder in the samples directory. Copy that complete folder to `/var/www`.*

Your Notes:

Exercise 3.7: Rules & Chaining of Rules

With the `chain` action, we can join rules to form more complex conditions.

1. Create a chained rule, which blocks request that contain a parameter `variable` but do not contain the parameter `value`.
2. Extend your rule above to only match for POST requests.

Your Notes:

Exercise 3.8: Flow Control: Skipping Rules

Using the `skip` and `skipAfter` actions, we can create conditional jumps in our rules.

1. Create a rule that blocks if the parameter `value` contains the word `test`. Use the `id` action to set the ID for that rule to `1000`.
2. Insert another rule before the previously created rule, which will skip the rule 1000 if the `REQUEST_METHOD` is `GET`. Note that this rule should not block!
3. Reload Apache and test your rule using the demo application.

Skipping rules can lead to problems, if another configuration removed a rule (i.e. due to eliminating false positives). The use of *markers* is an alternative:

1. Add a `SecMarker` directive after the blocking rule 1000 and change `skip` to `skipAfter` with the defined marker.
2. Reload Apache and verify the effect is the same. Check the debug log.

Your Notes:

Exercise 3.9: Using environment variables

The `setenv` action is a very important one as it allows you to set environment variables during request processing. These variables request in the Apache request processing scope and can be accessed by oder modules.

1. Add a new `CustomLog` directive to your virtual host:

   ```
   # conditional logging
   CustomLog "/var/log/apache2/post.log" combined env=myLog
   ```

2. Add a ModSecurity rule which will cause requests to the demo application to be logged, if the request is a POST request.

3. Extend this rule to log POST requests with large request bodies only, where *large* is everything larger than 128 bytes.

Hint: *The ModSecurity variable* ***`REQUEST_HEADERS:Content-Length`*** *contains the value of the* ***`Content-Length`*** *header.*

Your Notes:

Exercise 3.10: Environment Variables and other Modules

A very helpful module to use is the `mod_headers` module, which allows for the addition and removal of headers:

```
# mod_headers example: #
#  add the Response Header "X-MyScore: MYSCORE"
#  where MYSCORE is replaced by the environment
#  variable MYSCORE
#
Header add X-MyScore "%{MYSCORE}e"

#  similar for request headers:
#
RequestHeader add X-MyScore "%{MYSCORE}e"
```

1. Use the `Header` directive from above to propagate some data from within ModSecurity to the client. As an example, add a header `X-MyMethod` which contains the request method.

2. Check the audit-log file for the result of your experiments.

3. Add other ModSecurity variables such as the `UNIQUE_ID` to the response headers and test your results.

Your Notes:

Exercise 3.11: `exec` action: Calling external Scripts

Another useful case for environment variables is the execution of external scripts. This can for example be used to call `iptables` and insert a client IP into a predefined chain for blocking or any other system action.
The `exec:/path/to/script` action can be used to call external scripts. These will be called with no arguments, but the environment variables of Apache will be accessible by the script.

1. Create a small perl script `env.pl` like the following one:

    ```
    #!/usr/bin/perl -w
    #
    open (OUT, ">> /tmp/env.dat") || die "0 Error";

    foreach $key (keys %ENV) {
      print OUT "$key = $ENV{$key}\n"
    }
    #print "1";
    ```

    and place that script in `/opt/modsecurity/bin`. Change that script to be executable and ensure that user `www-data` can access and execute it.

2. Create a rule that will execute that script for POST requests to the `/objects.jsp` URL of the demo application.

3. Trigger that rule by accessing the demo application and check the `env.dat` file. Does it exist? Did something go wrong?

4. Check the debug-log of ModSecurity for the errors.

5. Transfer more information from ModSecurity into the `env.dat` file using the `env.pl` script and the `setenv` action.

Your Notes:

Exercise 3.12: Rule Order: Following the Rule Processing

Use the following set of rules and add them to your setup. Check the error log to see in which order they have been processed.

```
#
SecDefaultAction phase:2,pass,log

SecRule REQUEST_URI log,msg:'Found Request URI',id:1001

<Location /index.jsp>
   SecRule REQUEST_METHOD "@eq GET" "phase:1,msg:'GET req',id:1002"
   SecRule &ARGS "@ge 0" phase:2,msg:'ARGS?',id:1003
   SecAction msg:'Access to index.jsp'
</Loction>

SecRule &ARGS "@le 9" "phase:1,msg:'less than 9 ARGS?',id:1005"
```

Hint: *To safe you from typing, this snippet is provided in the* `samples` *directory as file* `rule-order.conf`*.*

Your Notes:

Exercise 3.13: Rule Inheritance: SecRule and Containers

An important issue to rule writing is the definition of rules in different scopes of your Apache server. We already noted the scope of directives during the setup of ModSecurity. Usually, all nested containers such as VirtualHost, Directory, Location, etc. will inherit all rules that have been defined in their parent container.
The SecRuleInheritance can be used to turn that behaviour off.

1. Disable the rule inheritance for the virtual host listening on port 81.
2. Verify that none of the global rules is applied to requests to that virtual host.

Your Notes:

Exercise 3.14: Adjusting Rules: Removing rules with `SecRuleRemoveById`

Instead of disabling the inheritance of rules completely for a container, we might sometimes only want to remove specific rules. We can do this by using the `ctl` action or `SecRuleRemoveById`.

1. Within the virtual host of port 81, remove a single rule by its ID using the `SecRuleRemoveById` command. Also try removing a range of IDs.
2. Verify that these rules are not processed anymore by checking the debug log.
3. Use the `ctl:ruleRemoveById` action to remove one of your rules for the `/objects.jsp` URL of the virtual host for port 81.
4. Verify the removal of that rule by trying to trigger the rule and following th debug log.

Your Notes:

Exercise 3.15: Changing Rules

Sometimes, some specific parameters need to be excluded from rule processing as they produce false positives. The `SecRuleUpdateTargetById` command can be used to fix that:

```
# Remove parameter 'text' from rule 1234
SecRuleUpdateTargetById 1234 ARGS:!text
```

1. Create a rule that checks all request parameters and verify your rule with the *sink* demo application.
2. Update that rule to exclude the `variable` parameter from the rule check and verify your new configuration.
3. Use the `SecRuleUpdateActionById` command to udpate the actions of your rule to block requests with status `409`.

Your Notes:

4. LogManagement with the AuditConsole

Exercise 4.1: Installing the jwall.org AuditConsole

In this exercise we will install and setup the AuditConsole application using the standalone-zip package provided in the `jwall.org` directory of the virtual machine.

1. Unpack the file `AuditConsole-0.4.6-9-standalone.zip` in the `/opt` directory and run

   ```
   $ chmod 755 /opt/AuditConsole/bin/*.sh
   ```

2. Before starting the AuditConsole, check the Tomcat configuration file

   ```
   /opt/AuditConsole/conf/server.xml
   ```

   as the default port 8080 is already in use by the demo application. Change that port to some other, e.g. 7080.

3. Create a MySQL database `AuditConsoleDB` and make it accessible by user ”console” from within the localhost.

   ```
   $ sudo mysqladmin create AuditConsoleDB
   $ mysql AuditConsoleDB
   mysql> GRANT ALL ON AuditConsoleDB.* to console@localhost \
   identified by ’console’
   mysql> flush privileges;
   ```

4. Check accessibilty of the database by running

   ```
   $ mysql -u console -h localhost AuditConsoleDB -p
   ```

5. Start the AuditConsole and open the web-interface at the port you used (e.g. 7080). Log in as user `admin` with password `admin` and follow the setup wizzard.

6. Create a sensor `sensor` with password `test` and use the `jwall-tools` to send your audit-log data from the first day to the AuditConsole.

Your Notes:

Exercise 4.2: Sending serial audit-logs using the jwall-tools

A very easy and quick way to send serial audit-log data to the AuditConsole is by using the `send` command of the *jwall-tools*:

```
$ jwall send http://sensor:test@localhost:7080/rpc/auditLogReceiver \
   /path/to/audit.log
```

1. Log into the virtual machine and locate the audit-log file.
2. Use the `jwall send` command from above to send the audit-logs from last day to the AuditConsole.
3. Open up the AuditConsole in your browser, switch to the *Event Browser* and hit the *Reload* button.

Your Notes:

Exercise 4.3: Remote Logging to the AuditConsole

In order to constantly send transaction logs to the AuditConsole in real-time, ModSecurity ships the `mlogc` tool. This will listen for transaction log summaries created by ModSecurity in `Concurrent` audit-log mode and send the events to a console server via `http` or `https`.

1. Switch the audit-log settings of your ModSecurity to `Concurrent` logging. Adjust the `SecAuditLogStorageDirectory` to the location where you want to have ModSecurity log to.
2. Restart Apache, access the demo application and verify that transactions get logged in concurrent mode, now.
3. Copy the `mlogc.conf` template from the `samples` directory to `/opt/modsecurity/etc/`.
4. Adjust the `mlogc.conf` to match your ModSecurity log settings.
5. Restart Apache and try to trigger events to be sent to the AuditConsole.

Your Notes:

Exercise 4.4: Event Filtering and Analysis

The most important feature of the AuditConsole is its filtering capabilities for AuditEvents. Some of the filters can simply be created by using the context-menu on event properties.

1. Create a filter to show all POST events that have been received by the AuditConsole.
2. Add another filter condition to only view POST requests to `/objects.jsp`.
3. Use the *Edit* button to specify a more complex filter manually.
4. Check the detailed event view and expand the *Rules Section*. Check which rules have been reported by ModSecurity.

Your Notes:

Exercise 4.5: Tagging Events manually and by Event Rules

Events within the AuditConsole can be tagged with custom tags. These can be used for filtering and may also be used within report creation.

1. Manually tag some events with the string *ignore*. Select all events tagged as *ignore* with a filter and use the *Delete* button to delete the filtered events.

2. Manually tag some events with the string *test*. Create a filter for the tag *test* and download the events selected by that filter.

3. Create another filter condition for events which do not have the method GET or POST. Use the *Create Rule* button to create a new rule for these events, which tags these events as *mostly harmless*.

Your Notes:

Exercise 4.6: Creating Access-/Error-Log Observers (optional)

Starting with the latest 0.4.3 version, the AuditConsole provides parsers for Apache *access-* and *error-log* files. By creating a simple *observer thread* within the AuditConsole it is possible to constantly read local files and store them in the database.
This allows to easily search/filter messages from plain one-line Apache log files.

1. Open the *System/Sensors* view from the toolbar of the AuditConsole and select *File Observers*. Use the *Add Observer* button to create a new file observer. Select the log-file you want to observe using the file-chooser dialog and choose the correct file format. Optionally, choose a *site* and/or *sensor* to associate events from that file with.

2. Check the *Events/Log-Messages* view from the toolbar and use the *Reload* button to reload the table.

3. Try filtering for events related to POST requests.

Your Notes:

5. Advanced Rules

Exercise 5.1: Using Collections

The most basic collection is the TX collection, which has the lifetime scope of a single transaction. It can be used to set flags which can be checked for by subsequent rules. Create a set of 3 rules, which act as follows:

1. Initialize the TX collection using the REMOTE_ADDR variable.
2. Set the variable get_method to 1 if the request is a get request.
3. Create a second rule which logs the transaction if the variable get_method is larger than 0. The log message shall include the variable's value.

Your Notes:

Exercise 5.2: Persistent Collections

In this exercise we want to create a simple rule that checks whether a client has sent POST request in the past. Clients should be tracked by their *User-Agent* string. If a client does not provide a user agent, the request should be blocked.

1. Create a (set of) rule(s) to check whether a client has accessed the demo application with a POST request.
2. Extend your rule to add an additional cookie called `POST_ACCESS` that is sent to the client.
3. Verify your rules using the demo application and the debug-log. You may want to use the *AuditViewer* a browser with a different user-agent string for testing.

Hint: *The* jwall-tools *do provide a* ***`collections`*** *command, which can be used to monitor the state of ModSecurity collections. Use*

```
$ jwall collections -r -v /opt/modsecurity/var/data
```

in a separate terminal to check the state of your collections.

Your Notes:

Exercise 5.3: Detecting changing User-Agents

With the prerequisites of the previous exercise we are now able to detect changes of the user-agent for an IP address.

1. Before starting to write a rule, make a sketch representing the sequence that you want to observe and what information is available.
2. Write down your assertion algorithm in pseudo code and then transfer it into a ModSecurity rule.
3. When the rule is in place, restart Apache and access the demo application to trigger the rule.

Comment: *This is of course more of artificial nature as several users might access a server from behind an application and use different browsers for that.*

Your Notes:

Exercise 5.4: Detecting Session Fixation Attacks

Tracking sessions is a bit more complex. Again, start as simple as possible and make a sketch on how you want to detect session fixation:

- How is a session handled in HTTP?
- How is that processes violated during a session fixation attack?

Write down your assertion/detection mechanism in pseudo-code and after that implement it as a ModSecurity rule:

1. The demo application uses `JSESSIONID` as name for the session identifier.
2. Implement your session-fixation detection in ModSecurity.
3. Open up the AuditViewer and load a serial audit-log file. Use the re-injection function to re-inject a request. Add an invalid session-identifier to that request and ensure that your rules log the detection of this invalid session.

Your Notes:

Exercise 5.5: Anomaly Scoring

One way of writing rules is the concept of *anomaly scoring*, where each detection mechanism adds a score to the anomaly score of the current request. A final rule then checks the score and blocks if this score exceeds a threshold.

1. Adjust all previous rules to add a score value to the `TX:SCORE` variable.
2. Create a new rule that manages a session score in `SESSION:SCORE`, which holds the sum of all request scores for a user session.
3. Create rules which block a request if the `TX:SCORE` exceeds a predefined transaction oriented threshold or the `SESSION:SCORE` exceeds a threshold defined for sessions.
4. Ensure that your rules properly log malicious requests, including the scores which lead to their denial.

Your Notes:

Exercise 5.6: Limit Requests per IP

In this exercise we want to block users based on their IP address after they have exceeded a specified number of requests within one minute.

1. Count the number of requests for an IP address
2. Block the address if the count exceeds 5 by responding with an error status of 505.
3. The block should expire after 60 seconds.

Your Notes:

Exercise 5.7: A simple Rule using Lua

Going back to exercise 16 we now try to achieve the same functionality using a Lua script and the `SecRuleScript` directive. This time we're looking for the `variable` parameter and want the rule to match, if that variable equals the value `admin`.

1. Start with the empty Lua script from the slide that never matches.
2. Extend the main function to retrieve the `variable` parameter from the request and test it against the string `admin`.
3. Return an appropriate value.
4. Create a `SecRuleScript` rule that uses your Lua function to check the `variable` parameter. The rule shall execute the `env.pl` script from exercise 3.11.
5. Test your rule with the demo application and check the debug-log.

Your Notes:

Exercise 5.8: Lua: Blocking with iptables

If we encounter a client IP that exceeds a request limit, we might want to completely block that IP at the firewall level. There are several ways to do this from within ModSecurity, e.g. using the `exec` action for calling a script or using Lua.
In this exercise we'll concentrate on Lua.

1. Create a small shell script `/opt/modsecurity/bin/blacklist` which takes a single IP address as parameter and injects that address into `iptables`.
2. First create a rule that will manage calling that script with the `exec` action. Remember that `exec` does not allow specifying arguments to scripts.
3. Create a Lua function according to the example presented on the slides to do the call with arguments.

Your Notes:

Exercise 5.9: Using `@rbl` with the jwall.org AuditConsole

The objective of this exercise is to setup the AuditConsole RBL server and use the `@rbl` operator to block clients that are on the block list.

1. Log into the AuditConsole and start the *RBL Service* in the *System* → *Setup* menu.
2. Edit the `/etc/dnsmasq.conf` configuration file of the DNSmasq server and add a route for our `rbl.localnet` domain:

   ```
   # send '.rbl.localnet' queries to the AuditConsole RBL
   server=/rbl.localnet/127.0.0.1#15353
   ```

3. Restart the DNSmasq server:

   ```
   $ sudo /etc/init.d/dnsmasq restart
   ```

4. Create a ModSecurity rule to query the RBL and test your rule.

Your Notes:

6. ModSecurity Core Rules

Exercise 6.10: Installing the ModSecurity Core-Rules

The task of this exercise is to download and install the *ModSecurity Core Rules*. The rules can be found at `modsecurity.org` or in the homedirectory of the user `modsecurity`.

1. Install the ModSecurity Core Rules into your ModSecurity directory.
2. Include the Rules within your ModSecurity configuration.
3. Access the Apache with your browser and try to trigger an alert.

Your Notes:

Exercise 6.11: Measuring Performance (Again)

We measured performance of the Apache and ModSecurity setup using different settings of debug log before. Back then, we did not have a lot of rules installed so there was not much logging involved anyway.
This is different now.

1. Ensure that the ModSecurity rules are included in your setup.
2. Raise the `SecDebugLogLevel` to various values (3-9) and run the *jwall-tools*' `eval` command with the test-payload.
3. Compare the different request rates with debugging enabled and disabled.
4. Remove the inclusion of the *Core Rules* and inject the payload again.

Your Notes:

Exercise 6.12: Using the Core Rules for the AuditConsole

The *ModSecurity Core Rules* are intended to provide generic attack detection. However, since the application level security is a complex issue, false positives will occur and need to be handled.
In this exercise we will go through the process of identifying false alerts and creating exceptions for these.

1. Set up a virtual-host listening on port 82 which should serve as reverse proxy for the AuditConsole.

2. Create a new sensor in the AuditConsole and set up remote-logging in this new virtual host to send the events to the AuditConsole.

3. Access the AuditConsole via the new reverse proxy and check the alerts generated by that.

Your Notes:

Your Notes:

A. Virtual Machine

The virtual machine provided for this workshop provides a standard Linux system for excercises. It will serve as a local system for setting up a ModSecurity enabled Apache Reverse-Proxy server. This section documents the process how the machine has been created and what packages have been added.

Creation of the Virtual Machine Image

The virtual machine image is a standard Ubuntu 13.04 64-Bit installation.
The installation has been run with only the bare basic packages having been installed, first. Additionally, the following packages have been added:

1. *Open SSH Server*

   Added by issuing `apt-get install openssh-server`. For speeding up the login, the line

   ```
   UseDNS no
   ```

   has been added in `/etc/ssh/sshd_config`.

2. *Java Runtime Environment*

   Added by issuing:

   ```
   $ sudo apt-get install openjdk-7-jre
   ```

3. *Apache Webserver*

   Apache has been installed issuing:

   ```
   $ apt-get install apache2
   ```

4. *Development Packages*

   In order to install the ModSecurity module from source, the Apache development packages and some additional library-header files have been added by issuing:

   ```
   $ apt-get install apache2-threaded-dev libxml2-dev
   liblua5.1-dev
   $ apt-get install libcurl4-openssl-dev
   ```

5. *MySQL Database*

   ```
   $ sudo apt-get install mysql-server
   ```

To reduce the file-system size the downloaded packages have been removed by running the command `apt-get clean`.

Accessing the System

An account for user `modsecurity` with password `modsecurity` has been set up in the system. This user has *sudo*-rights.