
Portable Protection Recipes for Web Application Defence

Table of Contents

Minimising the window of opportunity	2
The Impact of Portability on Protection	3
Software vendor supplied recipes	3
Third-party recipes	3
Recipes written by hand	3
Automatically created recipes	4
Portable Protection Recipes Format	4
Communication Protocols	6
Talking to recipe repositories	6
Talking to protection devices	8
Reference	9
Recipes	9
Rule sets	9
Rules	10
Recipe Metadata	10
Object model	12
Processing stages	13
Operators	14
Normalisation functions	15
Actions	15
A. XML Schemas	16
Recipe	16
Recipe metadata	16
Recipe list	16
B. Examples	16
C. List of possible improvements	16

<authorblurb>

Note: This is a rough draft of the proposed format. The technical aspect of the document is much better. However, there is still a long way to go with regards to document structure and the language. Please try to disregard non-technical issues for the time being. -- Ivan Ristic

</authorblurb>

Abstract

This document proposes a new, portable, format for web application firewall rule storage and transport. On the one hand, we want the end users to be able to learn one language for the majority of their tasks with web application firewalls. On the other, we want to allow the interested parties to create and freely exchange the knowledge how web applications can be protected from known vulnerabilities.

Minimising the window of opportunity

As organisations embrace web applications for their business advantage, they also become exposed to many security problems common. Programming errors in web applications often translate directly into vulnerabilities. Such vulnerabilities are frequently very easy to find and exploit. It is often said web applications help organisations expand their reach. This is true. But in the same manner, the vulnerabilities too will be exposed to a much wider audience.

The correct way to address vulnerabilities is by employing preventive measures during the software design and programming phases, taking all possible measures to make sure security-sensitive programming errors are not created in the first place. But, ultimately, 100% security is not possible to achieve. Therefore, the correct approach for organisations with regards to web application vulnerabilities is to assume they will happen and to adopt strategies to deal with them when they do happen.

A preferred way to deal with a vulnerability is to fix the problem in the application code. While one must work to deploy the fix as soon as possible, in reality, however, there are many factors will be delaying the resolution:

- The source code may not be available if someone else developed the application. This is the case with most off-the-shelf applications. In such cases the best one can do is report the problem to the vendor and wait for the fix to arrive. When and if the fix arrives, it can often be applied only to the most recent version of the application. Therefore in order to fix the vulnerability one must upgrade the application first. The upgrade is often not a trivial matter, and it may require careful planning and preparation, all of which takes time.
- If the problem is in a bespoke application developed by a third-party there may be legal reasons for not being able to touch the source code, even when it is available.
- For applications that are developed in house, a lack of resources sometimes stands in the way of action. Developers may be engaged with a more pressing matter, or it may simply happen that the key personnel are not present at the moment.
- Finally, if the problem is in a legacy application, changing the code to fix the problem may be possible in theory. But in practice there is often no one who knows how to do it, or the change may be considered too dangerous due to the lack of understanding of how the legacy system works.

Even under perfect circumstances there will be a delay between the time the problem is discovered and the time the problem is fixed. This delay is often measured in weeks. For example, there are strict procedures

to follow when complex applications need to be changed, and installations updated. The bigger the application the longer developers need to work to make sure the changes are safely implemented. But the longer problem remains unfixed, the greater the chances of it being exploited.

One way to deal with this issue is to deploy a Web Application Firewall (WAF) as a temporary protective measure. Most end users are familiar with network firewalls - they are quite widespread. In fact, it is impossible to imagine a well-designed network architecture without one or more network firewalls. Web application firewalls are similar in nature to network firewalls, but they operate on a higher level. While network firewalls concern themselves with raw network packets, web application firewalls operate with the high-level HTTP protocol. As such, most WAFs have advanced protective capabilities: they can be configured to observe the traffic and protect applications from attacks. This unique feature makes web application firewalls a reliable external patching tool. (Of course, web application firewalls can do much more, but discussing other features is not within the scope of this document.) With a WAF in place, it becomes possible to fix a problem with no access to the application source code. Such fix can be deployed quickly, much faster than the actual fix to the source code. The end result is that the window of opportunity to exploit the problem is significantly reduced.

The Impact of Portability on Protection

The immediate goal of web application protection will typically be achieved with any web application firewall, irrespective of the way the end user interact with it. However, if a portable, automation-friendly, format for protection rules is adopted, and it gains momentum, some of the following use cases may become realistic.

Software vendor supplied recipes

In response to a discovered vulnerability software vendors will typically release a patch or an upgrade. Security-conscientious vendors will understand that not all users can upgrade immediately. In response to this they may decide to design a protection recipe to allow their users to be safe until they can perform the upgrade.

Third-party recipes

If there is sufficient from the end-users, third parties may decide to offer subscription-based protection for web applications, even in the cases where the vendors themselves are not producing the recipes. This concept extends the concept of vulnerability database that is in widespread use today.

Recipes written by hand

End users themselves will design protection recipes for the vulnerabilities discovered in their bespoke applications. The benefit of portability here comes from the fact that only one rule format needs to be learned for use with any compatible web application firewall.

Automatically created recipes

The potential for automation of protection is quite large. Here are some usage scenarios:

- Web vulnerability scanners are often used to interact with web applications in order to discover security problems. When a problem is discovered the vulnerability scanner can create a rule on the fly, and communicate with the protective device to initiate protection.
- If there is a web application firewall deployed and it knows which web applications are deployed, it can automatically download and deploy protection recipes from one or more recipe servers (e.g. internal recipe servers, vendor recipe servers, or servers managed by a trusted third party).
- An attack can be detected by any number of tools and devices. For example, Intrusion Detection Systems are often deployed to monitor networks. So are the scripts to monitor the web server logs. Even applications themselves can be configured to watch for attack attempts. These tools can all communicate with protective devices to install or at least suggest protective measures.

Portable Protection Recipes Format

This section introduces the portable protection rules format. We approached the format design with the following guidelines in mind:

1. The format must be very simple. We want people to be able to write recipes with their text editors with little, or very little need to read the documentation.
2. Make simple things simple, complex things possible.
3. The format must be designed so that is possible to process it efficiently. This basically ruled out any use of programming languages for recipes - such design would never have worked with busy web sites.
4. The format must fulfill its purpose--be able to protect web applications--but should not attempt to do anything more. First of all, it is our opinion that is not possible to create a portable format that would encompass complete WAF functionality. Furthermore, an attempt to do so would be doomed to failure for political reasons. In our opinion, a minimum-functionality format will not threaten to vendors. Such a format leaves plenty of room for vendors to compete, yet it makes interoperability possible.

Without further ado, let us examine a simple example. The following recipe will examine all parameters of every request and look for a keyword "bomb". It will stop request processing when the keyword is found.

```
<recipe>
  <ruleSet>
    <rule
      operator = "regex"
      arg = "params[*]"
      value = "bomb"
```

```
    />
  </ruleSet>
</recipe>
```

But let's assume there are certain parameters where it is allowed to have the keyword "bomb". The following recipe creates an exception; all parameters except parameter named "safe" will be examined for the keyword.

```
<recipe>
  <ruleSet>
    <rule
      arg = "params[*] !params['safe']"
      value = "bomb"
    />
  </ruleSet>
</recipe>
```

So far we have used only one rule set and one rule per recipe. The rule sets exist to enable recipe writers to tie several individual rules with an logical AND condition. This is exactly what the following example does. There are two rules in the rule set. Only if a match in *both* rules (meaning there is both "explosive" and "detonator" appear in request parameters) will cause the whole recipe to match (and a request to be rejected):

```
<recipe>
  <ruleSet
    condition = "and"
  >
    <rule
      arg = "params[*]"
      value = "explosive"
    />
    <rule
      arg = "params[*]"
      value = "detonator"
    />
  </ruleSet>
</recipe>
```

The format supports many other optional features. Some of them are:

- By default, rules use regular expressions for comparison. Many other operators are also supported.
- Various anti-evasion measures can be configured and performed automatically.
- It is not necessary to interrupt request processing on rule match. A recipe can choose only to emit a warning.
- Rule sets can be configured to be run at different phases of the request. For example, it is possible to run some rule sets at the beginning (e.g. after only the headers have been read) or at the end.

Examine the Reference section later in the document for full information about what is supported.

Communication Protocols

A portable rule format solves only one half of the problem. To take the other half of the problem we need to design a communication protocol that would allow recipes to be exchanged. We decided to rely on the existing standards:

1. Use HTTP to transport recipes in request payloads
2. Use Basic authentication
3. To secure the communication channel use SSL
4. Use client certificates where a high level of security is required

The remainder of this section explains each message in detail.

Talking to recipe repositories

The following operations are supported:

- Retrieve recipe
- Create recipe
- Remove recipe
- Revoke/reinstate recipe
- Locate (search for) recipes in a variety of ways

Retrieving recipes

Each recipe is assigned a unique URI in the repository. The following examples all assume the repository root is located at "http://www.example.com/repository/". Recipes are always located in the "/recipes/" subfolder of the repository.

Use a GET request and the recipe URI to retrieve the recipe. The following will return the latest version of the recipe "ts001":

```
GET /repository/recipes/ts001 HTTP/1.0
Host: www.example.com
```

To retrieve a specific version, append the version number to the URI. The following will return version "3" of the recipe "ts001".

```
GET /repository/recipes/ts001/3 HTTP/1.0
Host: www.example.com
```

Creating recipes

To create a recipe use the same URI in a PUT request, supplying the recipe in the request body.

```
PUT /repository/recipes/ts001 HTTP/1.0
Host: www.example.com
Content-Length: 3489
```

...complete recipe in the request body...

It is possible but not necessary to append the version number to the URI. Repositories must not accept recipes with the same ID and version number as one that already exist. The repository must also check the recipe ID embedded in the request body matches the one used in the URI. If the version number is used in the URI it too must match the version number in the recipe. After the new recipe is accepted, any existing versions should be marked obsolete.

Removing recipes

It is generally not recommended to remove recipes from repositories. This option should be used only as a last resort, e.g. if the recipe was published by mistake. If the recipe has been distributed then it must be revoked in order to notify the users. Use a DELETE request to delete a recipe:

```
DELETE /repository/recipes/ts001 HTTP/1.0
Host: www.example.com
```

Recipe revocation

If a recipe is found to be faulty it must be revoked. Merely deleting such recipe is not enough because those devices that may have deployed it will not be notified. To revoke a recipe a single POST request is required:

```
POST /repository/vokeRecipe HTTP/1.0
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 43
```

```
recipeid=ts001&version=3&revocationDate=XYZ
```

It is also possible to revert revocation:

```
POST /repository/unRevokeRecipe HTTP/1.0
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 24
```

```
recipeid=ts001&version=3
```

A faulty recipe must be revoked even in the cases when a newer (presumably correct) version is published. Publishers are expected to first upload the new recipe, and then revoke the previous version.

Finding recipes

The following search methods are supported (M stands for mandatory parameter):

- listRecipes(changedSince)
- listProductRecipes(productName M, productVersion, changedSince)
- listRecipeVersions(recipeId M)
- listPublisherRecipes(publisherId M)

The response body of each search request will contain a list (zero or more) of metaData elements of matching recipes:

```
<recipeList>

  <metaData>
    <!-- meta data -->
  </metaData>

  <metaData>
    <!-- meta data -->
  </metaData>

  <!-- Other meta data elements here -->

</recipeList>
```

Note: repositories must not use the releaseDate and revocationDate fields to compare with the changedSince parameter. They must keep their timestamps internally to allow the clients to retrieve the changes since the last time they contacted the repository.

Talking to protection devices

Protection devices are expected to implement the methods required for repositories. However, certain additional methods and parameters are required to achieve additional functionality. The following additional parameters are required for recipe creation:

- virtualHost - which virtual host to apply recipe to.
- path - path prefix, apply recipe only to the request that fall within this path. A recipe may be designed to work for a particular application, but the application may be installed at a subfolder on the web site and not directly in the root folder.
- duration - the duration, in seconds, to specify when should the recipe be removed.
- actionOverride - if the recipe is designed with prevention in mind, but some sites do not want to use

prevention.

One additional search function:

- `listSiteRecipes` (virtualHost M, path) - to find only the recipes that apply to given host and path.

Reference

This section contains a descriptive definition of the portable protection recipe format. You will find the schemas in Appendix A.

Recipes

A recipe consists of an optional meta data section followed by zero or more rule sets. Only the publishers are required to fill in the meta data section for the recipes they intend to publish. The end-users are allowed to concentrate of solving the immediate problem of deploying protection for their web applications.

To make recipe writing easier, recipes support the fields below as attributes. These attributes have no direct meaning for the `<recipe>` tag. Instead, they are used as default values for the rule sets and rules in the recipe.

- `stage` - at which stage to run the rule set. The default is *requestBody*.
- `action` - what happens when a rule set matches. The default is *error*.
- `normalisation` - which anti-evasion techniques to apply to data automatically. The default is *none*.
- `path` - only requests matching this patch will be processed. The default is "".
- `message` - the message to output on rule set match. The default message is "TODO".

Rule sets

A rule set is a collection of rules. Rule sets support the same attributes already explained for the recipe tag:

- `stage`
- `action`
- `normalisation`
- `path`
- `message`

The following restrictions apply:

- The path given in a rule set cannot be less restrictive than a path given in a recipe.
- The normalisation attribute supports a relative format, which contains only the changes from the normalisation configuration inherited from the recipe (e.g. "+decodeURLEncoded" means to add this anti-evasion technique to the list of techniques already specified in the recipe tag. If a relative configuration format is used, each normalisation technique must be preceded by either a "+" or a "-" character.

A special attribute can be used to specify whether the rules in the rule set are to be treated as one logical rule (AND), or as separate rules (OR):

- condition (default is OR)

Rules

Rules can have the following attributes:

- normalisation (absolute or relative format, default *none*)
- operator (default *regex*)
- args - mandatory field, it contains a list of field to apply the operation to
- value - mandatory field, it contains the second parameter for the operation

Recipe Metadata

Recipe writers fall into one of two categories: those who write recipes for their own use, and those with intention to publish recipes to other users. The former group will typically not be interested in supplying additional meta data for their recipes. Publishers, however, need to make additional information available, to allow end users to find the recipes using a search mechanism and to determine whether recipes are applicable to their installations.

```
<recipe>
  <metadata>

    <!-- Recipes that are to be published must include the
         meta data (including the information about the publisher). -->

    <recipeId>ts001</recipeId>
    <status>VALID</status>
    <title>SQL injection in VulnerableProduct &lt;= 4.6.12</title>
    <version>1</version>
    <releaseDate>June 5, 2005 TODO(format)</releaseDate>
    <description>...</description>
    <licence>...</licence>
```

```
<licenceUri>...</licenceUri>

<publisher>
  <publisherId>ts</publisherId>
  <name>Thinking Stone</name>
  <uri>http://www.thinkingstone.com</uri>
  <email>pepperRecipes@thinkingstone.com</email>
  <repositoryUri>http://www.thinkingstone.com/repository/</repositoryUri>
</publisher>

<!-- Product information is mandatory if the recipe is designed
      with a specific product problem in mind. -->

<product>
  <name>VulnerableProduct</name>
  <vendor>VulnerableProductVendor</vendor>
  <productUri>http://www.vulnerableproductvendor.com/vulnerableProduct/</productUri>
  <vendorUri>http://www.vulnerableproductvendor.com/</vendorUri>
  <applicableVersions>
    <applicableVersion operator="lte" version="4.6.12"/>
  </applicableVersions>
</product>

<!-- References are optional, but we recommend them highly because they allow
      the end users to learn more about the problem. -->

<references>
  <reference>
    <title>SQL injection in VulnerableProduct &lt;= 4.6.12</title>
    <uri>http://seclists.org/lists/bugtraq/2005/Jun/9090909.html</uri>
    <type>advisory</type>

  </reference>
  <reference>
    <title>Minor security vulnerability discovered in VulnerableProduct</title>
    <uri>http://www.vulnerableproductvendor.com/vulnerableProduct/security2.html</uri>
    <type>vendorReport</type>
  </reference>
</references>

</metadata>

<!-- the remainder of the recipe omitted -->

</recipe>
```

The following fields can be used to construct recipe metadata:

- *recipeId* - unique recipe ID. To ensure uniqueness, recipe IDs must start with the unique publisher ID, and followed by any string selected by the publisher. The following recipe IDs are all valid: "ts001", "ts-1", "ts-001", "ts-20050603-1".
- *status* - one of VALID, OBSOLETE, REVOKED.

- *title* - human readable description of the recipe, up to 128 bytes in length.
- *version* - recipe version serial number. Serial number for a recipe always starts with 1, and increases by one with every subsequent version.
- *releaseDate* - date when recipe was published.
- *revocationDate* - date when the recipe was revoked or made obsolete.
- *description* - human readable description of what recipe does, up to 8192 bytes in length.
- *licence* - one of REDISTRIBUTION_ALLOWED, NONCOMMERCIAL_REDISTRIBUTION_ALLOWED, REDISTRIB_NOT_ALLOWED, OTHER.
- *licenceUri* - URI to a resource where full text of the licence can be found.

With exception of revocationDate, all fields are required for a repository to accept the recipe.

Publisher information

Publisher ID, name, URI, email, repository URI.

Product information

Product name, vendor name, product URI, vendor URI, applicable versions. Operators eq, lt, gt, lte, and gte are used to define the applicable versions.

References

Zero or more references can be attached to a recipe. References should be added to recipes to help end-users learn more about the problem. Each reference consists of three fields: title, type (advisory, vendor, bugtraq, cve, osvdb, other), and the URI.

Object model

```
request {
  server_name
  server_port
  server_protocol
  remote_addr
  remote_host
  remote_user
  remote_port
  content_length
  content_type
  query_string
  path_translated
  path_info
}
```

```
request_method
request_line
auth_type
script_name
request_uri
session_id
raw_body

params {
    name
    value
}

headers {
    name
    value
}

cookies {
    name
    value
}

files {
    name
    size
    tmp_filename
}
}

response {
    status_line
    status
    status_message
    content_length
    content_type
    raw_body

    headers {
        name
        value
    }

    cookies {
        name
        value
        // TODO more fields missing here
    }
}
```

Processing stages

There are four processing stages:

1. *requestHeaders* - occurs after the request line and the headers have been read.
2. *requestBody* - occurs after the request body has been read. This stage will occur even for requests that do not have bodies. It will occur even in the cases when the WAF is not configured to observe request bodies.
3. *responseHeaders* - occurs just before response headers are sent to client.
4. *responseBody* - occurs just before response body is sent to client. This stage will also occur in the cases when the response body is not available. (The response status and the headers are always available.)

The *requestBody* and *responseBody* stages should be viewed as main processing stages. The earlier stages (*requestHeaders* and *responseHeaders*) exist only to those authors who have a need to optimise their recipes, for example, to perform an operation before a request body is read. Recipe writers who choose to use the earlier processing stages should note that at that point only partial information will be available to the rule. For example:

- Parameters transported in the request body (e.g. POST) will become known only after the body is read.
- It is possible to transport additional request headers after the body.

Operators

The available operators are listed below:

- *regex*
- *nregex* (not *regex*)
- *lt* (less than)
- *gt* (greater than)
- *exists* (checks whether the named variable exists)
- *nexists* (not exists)
- *strstr* (string match)
- *eq* (equal)
- *neq* (not eq)
- *gte* (greater than or equal)

- `lte` (less than or equal)
- `size` (calculates size of each element)
- `count` (counts the number of elements, i.e. in the `args` attribute of a rule)
- `ipmatch` (matches an IP address)
- `nipmatch` (not `ipmatch`)
- `empty` (true if the variable does not exist, or if it is empty)
- `nempty` (not empty)

Normalisation functions

To counter evasion attempts the following functions can be used to pre-process data:

- `decodeURLEncoded` (e.g. convert `"%64%61%74%61"` to `"data"`)
- `decodeURLEncodedAgain` (for cases where double URL decoding is required)
- `decodeEscapeSequences` (e.g. convert `"\data"` to `"data"`)
- `compressWhitespaces` (e.g. convert `"DELETE FROM table"` to `"DELETE FROM table"`)
- `compressSlashes` (e.g. convert `"/path//to///folder"` to `"/path/to/folder"`)
- `convertBackslashes` (e.g. convert `"/path\to\folder"` to `"/path/to/folder"`)
- `removeSelfReferences` (e.g. convert `"/path./.to/folder"` to `"/path/to/folder"`)
- `convertToLowercase` (e.g. convert `"DeLEtE FroM table"` to `"delete from table"`)
- `none`

Actions

Every recipe can choose to perform one of the following actions:

1. *error* - submit an error to the engine. Upon receiving the error the engine should reject the current transaction, but only if it is configured in the prevention mode. If the engine chooses not to reject the transaction recipe processing must continue.
2. *warning* - report on suspicious activity. The engine will typically log the warning to the log. Recipe

processing will continue with other rule sets if they exist.

3. *notice* - submit a notice to the engine. The engine will typically log the notice to the log. Recipe processing will continue with other rule sets if they exist.
4. *allowRequest* - rule processing will stop. If there are rule sets that operate at responseHeaders and responseBody stages they will be processed. If there is a message associated with this action it must be logged at level notice.
5. *allowRecipe* - stop processing the recipe. If a message is associated with this action it must be logged as notice. None of the other rule sets will be processed in the subsequent processing stages.

The allow actions are designed to work only within the recipe they are used within. They must not influence other recipes in any way.

XML Schemas

Recipe

TODO

Recipe metadata

TODO

Recipe list

TODO

Examples

Example 1:

TODO

Example 2:

TODO

List of possible improvements

These improvements are still being considered:

- Add the ability to parse the URI into parameters. For example, the following URI `http://www.amazon.co.uk/exec/obidos/ASIN/0596007248/` contains an effective parameter "productid" whose value is "0596007248".
- Add functions to the format. One use for functions is to perform validation. For example, is this string properly URL-encoded?
- Consider possible inclusion of programming logic in JavaScript.