Enough with *Default Allow* in Web Applications!

Ivan Ristic
Vice President, Security Research

Ofer Shezaf
Vice President, Product Management

Breach Security Labs
Breach Security, Inc.

August 2008

ABSTRACT

The ***default allow*** deployment model used by web applications is the cause of numerous security problems—it forces applications to accept any requests, rather than only those they can handle. We propose that web applications adopt a ***default deny*** model instead, removing several classes of vulnerabilities and significantly reducing the attack surface for many others. Our approach works best when adopted during application development, but can be nearly as efficient in other phases of the software development lifecycle or during deployment.

INTRODUCTION

Everyone agrees that we have a terrible problem with the security of web applications. What got us into this mess is the organic growth of Internet technologies over many years, which happened with little thought about security. What was supposed to be a simple mechanism for document exchange exploded into an application delivery platform. Web applications are everywhere and we now find ourselves wishing we could go back in time to put things right. Instead, we must expend enormous effort just to make the present bearable. Our hands are tied: improvements can only be made incrementally, as we must keep the applications running while we improve them. Our best chance may be to slowly migrate to new platforms that are secure, while doing what we can to survive the current period of insecurity. This paper will outline a way we can do *both*.

Most of the time spent on web application security is used to uncover problems, improve coding practices and fix issues. Although these activities are unavoidable today, more time should be spent changing the way applications are developed and deployed, to systematically eliminate classes of problems. The only way to truly prevent security issues is to make sure they cannot be created in the first place. In other words, web application developers must make it difficult—hopefully impossible—to shoot themselves in the foot.

GOALS

Our goal with this paper is to change one of the very significant causes of web application insecurity—*the **default allow principle***. We object to the current common practice of web servers designed and configured to pass all requests to web applications for processing with little or no restrictions. That is in direct opposition to what security professionals consider as good practice.

Two current development practices further amplify the problem:

1. In developing web applications, programmers are forced to interface directly with many different protocols and specifications and all of their complexities. We believe this is too much to ask, considering the current state of software development where the focus is on delivering features with security as an afterthought. With new web security issues arising every day, even experts find it difficult to keep up. How can we expect programmers to do a good job?
2. In many cases, applications are built as simple collections of files, which web servers are instructed to process: scripts are executed; everything else is delivered verbatim. Thus, the file system becomes an implicit interface, but not many think of it that way. Simple omissions often escalate into vulnerabilities.

   For example, it is a common practice to have text editors preserve one previous file version in a backup file. Such backup files usually carry the same file name as the original, but use a different extension. If a backup file makes its way to the web server, the web server will not know that the file is not supposed to be there. Faced with an unknown extension, the web server is likely serve the file verbatim, thus causing an information leakage problem!

Although changing the way application development is done—by moving to libraries with higher abstraction levels—would result with better security overall, we feel that it is too late for that, at least in the short term. We cannot fight inertia. Given a choice, the majority of programmers would continue to develop in the way they are doing now. We feel that a mechanism that allows for gradual improvement, yet also works long term.

Benefits of a Default Allow Model

Decoupling of web applications from web servers can address the following web application security issues:

- Prevention of information leakage through files unintentionally included with applications.

- Mitigation of injection attacks[1].
- Elimination of injection attacks for certain parameter types (e.g. integers).
- Exploitation of so-called debug parameters, which were intended for debugging and troubleshooting, but were mistakenly left in production code.
- Exploitation of any functionality other than production code, even if it was left in the application inadvertently.
- Partial mitigation of buffer overflow and similar attacks by enforcing parameter size limits.
- Elimination of attacks that exploit errors in web server configuration (e.g. attempts to use **`PUT`**, **`DELETE`**, or any of the WebDAV methods).
- Reduction of the application attack surface by rejecting unknown content type encodings, and encodings that are not used by the application (e.g. do not allow requests using **`multipart/form-data`** if the application does not need it for file uploads).

Use Cases

Software Developers

Best results will be achieved if the ***default deny*** model is adopted in software development, with several options possible:

1. New applications can be written with ***default deny*** in mind, ensuring the model and the application are always in sync, because developers will need to declare every external function they wish exposed. Starting from scratch with ***default deny*** means the model will be updated incrementally, as changes to applications are made. This is easy to achieve.
2. Implementing ***default deny*** in existing applications is more difficult because developers require a solid understanding of the entire application to produce the entire model in a single attempt.
3. Even if ***default deny*** is not adopted in development, it can still be of significant benefit when it comes to fixing vulnerabilities quickly. Software developers can reduce the exposure window of their users by reacting to vulnerabilities quickly with virtual patches, while working in parallel to implement and test a proper fix.

Application Users[2]

Application users are sometimes forced to live with insecure applications. Faced with an application with a bad track record or with a vendor that is slow to react, users can help themselves by building and deploying ***default deny*** models on their own:[3]

1. Virtual patching is a popular and relatively simple way of reducing the window of opportunity for the attack. In anticipation of a proper fix from software manufacturers, users can write virtual patches using publicly-available vulnerability information, or review the application source code.

[1] Susceptibility to injection attacks is a result of missing or inadequate encoding of data at system boundaries. Input validation, the basis of our proposal, can only ensure data is in correct format. This has security benefits only if the format is strict enough to make injection attacks impossible. For example, it is very difficult to execute an injection attack when you are only able to use digits in attack payload.
[2] By "application users," we mean those in charge of installing and maintaining applications, not those accessing applications with their browsers.
[3] Not every user should be a web application security specialist. Those who can do this job themselves probably will; others can hire security consultants to do the work for them. In either case, the user is in control of the situation, which is what we desire.

2. Legacy applications are unlikely to get fixed, mostly because few understand how they work or dares to change them, for fear of breaking them. The use of such applications can be made bearable, even if they contain vulnerabilities, through deployment of a ***default deny*** security model.
3. Users of popular applications could collaborate to build models. In the case of open-source products, such efforts can even be spawned into community projects. A single high-traffic application deployment could use machine learning to arrive at a model that will then be distributed to all product users. Model deficiencies can be quickly resolved if many sites collaborate to build a single profile that works for all of them.

Web Application Firewalls

Web application firewalls are best suited to serve as enforcement points today when there is no support for ***default deny*** in the web application stack. They could be extended to support the import of application models, either manually or programmatically. In the former case, administrators could feed application models to web application firewalls as part of the deployment procedure. In the latter case, web vulnerability scanners could be configured to send virtual patches for every vulnerability they discover.

Furthermore, web application firewalls are in an ideal position to automatically generate positive security models—and enforce a ***default deny*** model—through observation of valid application traffic.

Web Vulnerability Scanners

Web vulnerability scanners typically perform two basic functions: application crawling and testing. In the crawling phase, the scanner works to discover the attack surface. This phase if very important: a missed resource will not be tested for security problems. Our proposal to use explicit application interfaces could be of great help to scanners as they could use the information to gain quick understanding of the application or to compare their understanding with the reality. A similar effect could be achieved even where there is no explicit application model, because a web application firewall could be deployed to build one.

Programmatic Interaction

Our research is currently focused on application modeling, but several of the proposed use cases ask for programmatic communication. Although models can be exchanged manually, the best effects will be achieved if tools can discover, retrieve and update models in an automated fashion. The communication protocols needed to achieve programmatic interaction are out of the focus of this paper, but we may address them at one point in the future. The reader is advised to read the Portable Web Application Firewall Rule Format[1] document, where programmatic interaction is covered in more detail.

PREVIOUS WORK

Default Deny

The idea of allowing only what is known and secure is not new. It has long been established as one of the cornerstones of good programming and good security. In spite of this knowledge, the ***default deny*** model is seldom found in practice.

Ranum[2] has an insightful account of how the ***default deny*** culture in security products lost to the *default allow* culture due to pressures for higher performance, lower cost and convenience. He writes:

> In the mid 1990's [*sic*] the author was selling proxy firewall products that had a superlative history of resisting attack; yet the market leading products were simplistic "stateful" packet filters that were sold based on the fact that they were faster, cheaper, and more forgiving. Put differently: they didn't perform as rigorous checks, so they could be fast. They were easier to code, so they were cheaper. They were more forgiving, because they were more permissive.

We believe it is safe to say that the above comment can be applied not only to network security, but to all our software development and application security practices today. The majority will do what is more convenient, rather than what is more secure. Checking of all program input is widely accepted as necessary, but programmers are consistently avoiding it, causing many of the application security problems.

For example, web applications typically use relational databases to store data. Database fields, which are used to store data, are almost always limited in size, but applications often do not check whether the user-provided data is within limits. This practice not only opens a door for exploitation (e.g. buffer overflows) but also propagates the problem, hides the root cause and results either in data truncation or database errors, depending on the database engine used[4] .

Related Work in the Web Application Space

Scott and Sharp[3] envisioned a Security Policy Description Language (SPDL), which essentially implemented a rudimentary application-level firewall with features such as input validation, input transformation, data signing, and support for negative security model in output.

The OWASP Stinger project[4] is a centralized input validation component for Java™ web applications, which can be used with both new and existing applications (without a need to change application code or have access to it) thanks to it being implemented as a Java Servlet Filter.

Struts[5] is a very popular framework for web application development in Java. It features an extendable validation framework based on XML, although much of the functionality is delegated to building blocks consisting of Java functions.

Kruegel and Vigna[6] wrote a very interesting paper on anomaly detection, which is similar in goal to that of Scott and Sharp, except that it uses statistics instead of heuristics to verify input data.

One of the authors designed a portable web application firewall rule format in 2005 along with a Java-based implementation, but the idea failed to take off.

ModSecurity™[7] is an open-source web application firewall that can work either embedded or as a network gateway. Its rule language is model-agnostic and supports both the ***default allow*** and ***default deny*** models. Although the current ModSecurity rule language can be used to construct solid positive security models, users would benefit from having a separate mechanism designed specifically for this task.

The REMO[8] project aims to make the process of writing positive security models easier by providing tool support, but it does not offer automation.

Christian Bockermann[9] started a project to provide an XML-based abstraction level for ModSecurity rules, which is especially useful when it comes to positive security models. A related tool called WebApplicationProfiler, also written by Bockermann, takes ModSecurity transaction logs (which contain full

[4] Some database engines will silently truncate data on input, themselves promoting the bad programming practice of hiding errors.

transaction content) on input and exports a positive security model. We are planning to use a similar approach for our proof of concept.

IMPLEMENTATION

To approach the problem, we take a view that the Internet is a computer, sites are programs, and each URL is a function call.[5] We basically treat HTTP as an API we can intercept, which is a view similar to that of AOP[11] programming. This position allows us to ignore application implementation details, supporting any standards-based web application platform. By identifying the basic building blocks of every web application, we arrive at an abstracted model that can be used to enforce the desired ***default deny*** mode of deployment.

Requirements

In this section, we describe the main requirements that guided us in designing our application modeling framework.

Portability

The format must be easy to consume on a wide variety of systems. This leads to the natural choice of XML for the storage format, which is universally supported and easy to parse.

Partial Model Support

To build a complete positive security model is only possible with simple applications or if the effort is automated or incorporated into the application development process. We feel we must support partial models for the following reasons:

- A person building a model is likely to work on one resource at a time. The entire process may last for an extended period of time, during which it would be advantageous to make deployment of partial models possible.
- Arriving to a partial model could be a goal of its own. Some users or application developers can simply decide to focus on the parts of an application that are more exposed than others. (For example, the attack on the login page that is exposed on the public Internet and well known is more likely to attract attacks than an internal function accessible only to a small number of users.)
- A partial model may be a goal on its own. This will be case with virtual patching, where the user sets out to fix a known application problem in the anticipation of a better fix in the code.
- Automated tools, on the other hand, are likely to work on the model for all resources without seeing many transactions for the same resource. Thus, partial model support here means we need to be ready to differentiate between the final version of the model, and the parts that are being built. We will use the term *confidence* to refer to partial model in this sense.

Real-Life Suitability

Although much of the infrastructure used to develop web applications is standardized, application developers do not always choose to use it in a standardized manner. Over the years, we have observed the infrastructure used and misused in ways not originally envisioned. A major requirement for us is the ability of the language to work with real-life applications, which often deviate from textbook examples.

[5] One of the authors still remembers when he first viewed the Internet in this light, while reading Philip and Alex's Guide to Web Publishing[10].

Ease-of-Use

We want to have a low barrier to entry, and to make it possible to write or update application profiles by hand. This requirement forced us to look away from using statistic in input validation. Statistics may work well for tools, but do not work as well with people.

Support for Non-Standard Behaviors

Parameterized URLs

Many applications will transport parameters not only in the usual places (i.e. the query string and request body), but embedded in their URLs as well. The Amazon URL http://www.amazon.com/dp/0596007248/ contains a book ISDN number. This transport method is not standardized, but the corresponding script on Amazon's servers knows how to extract and use such parameters. This technique is often used to make URLs easier to remember and use.

Resource Aliasing

One resource can sometimes appear under more than one URL. Requests for folders, for example, will force the web server to select a default resource to serve.

Gateway Pattern

While in some applications, one URL corresponds to one unit of work, many applications perform further request routing internally, based on parameters. Such applications may have two, three or more completely different request types (some applications are known to implement their entire functionality using a single gateway script) processed by what appears to be a single resource.

The most common variations of the gateway design pattern are:

- Request Method
  Applications will often implement two behaviors in a single resource: one that supports **`GET`** and the other that supports **`POST`**. The resource will most commonly respond to a **`GET`** by displaying a form, and responding to a **`POST`** by processing the supplied data.
- PATH_INFO
  `PATH_INFO` is the name of the variable in the CGI[12] specification that contains the part of the URL appended to the filename of the script. It is an extra bit of information a URL is sometimes allowed to contain.[6] Applications sometimes rely on the **`PATH_INFO`** information to implement external URLs that are user and search-engine friendly. Unless **`PATH_INFO`** is parameterized (e.g. to contain an ID of a product), however, this case does not necessarily need to be explicitly handled. From the outside, each URL will have its own behavior, which is exactly what we need.
- Command Parameter
  Some applications will have one parameter indicating the operation that needs to be processed. The name of such parameter is often ***`cmd`***, ***`command`*** or ***`action`***.
- Run-Time Parameters
  Some applications will generate parameters at run-time. This technique is quite common in PHP, which will automatically create arrays when presented with parameter names such as **`p[1]`**, **`p[2]`**, **`p[3]`** and so on.

[6] Some web servers allow it, some do not. Some can be configured to go either way.

Building Blocks

Application

In many instances, sites will contain one application, but this does not always have to be the case. Some sites can contain more than one application or even multiple instances of the same application.

Resource

A resource is a unit of work that handles requests. In many cases, one resource is equivalent to one script, although there are many cases where this is not true.

Resource Behavior

One unit of work can—and usually does—support multiple behaviors. The quality of application models will depend in large part on the ability to address each such behavior individually.

Parameter

In the end, applications must receive data. They accomplish this through parameters. In our model, each behavior can receive zero or more named parameters.

Parameter Attribute

Each parameter is modeled with a series of attributes, each of which addresses one aspect of a parameter. Cardinality, minimum length and maximum length are all examples of possible parameter attributes.

MODEL OVERVIEW

This section contains a description of the model through the storage format, using steps we envision will be taken to process each request. Some details have deliberately been omitted for clarity, as we wanted to focus on the design decisions. In particular, we do not mention metadata, but such information will be a required part of every application model.

Initialization

The main task of the initialization phase is to parse request parameters, but also to arrive at the effective URL, taking into consideration that an application can be configured to use any site path as its base URL. For example, a blog application can be installed directly onto a site blog.example.com or to a sub-path www.example.com/blog.

Resource Identification

An application is considered to be a collection of nested resources. The root resource must always be present and use the special name `/`. The format does not differentiate between folders and files, as such difference does not exist in the URL space. The following is an example of a simple website containing a few scripts:

```
<applicationModel>
   <resource name="/" alias="index.php" />
   <resource name="/index.php" />
   <resource name="/sign-in.php" />
   <resource name="/sign-out.php" />
   <resource name="/download.php" suffix="^/" />
</applicationModel>
```

Note:

- Although the resource paths appear as absolute in the model, they are really relative to the root of the application, which may not necessarily be the same as the root of a site. We appreciate that applications can be installed using different prefixes, and that there can even be many instances of the same application sharing the same domain name.
- A forward slash is the only acceptable path separator. Model verifiers and enforcers on platforms that support other path separators are advised to handle such cases with caution to prevent evasion.
- Resource names are case-sensitive. Case-sensitivity is a deployment configuration option outside the scope of the application model. Again, care should be taken to make sure no room for evasion exists.
- Note how one resource (**/**) was declared as an alias, mimicking the equivalent web server feature, which is in widespread use. Aliased resources are always validated using the information provided by the resource to which they point.
- We do not support resource nesting in the storage format. An earlier version of the design did, but we came to realize nesting would ultimately only make models difficult to read. Furthermore, one of our requirements was a support for virtual patching, which would be carried out with a partial application model. Nesting would be awkward to use in such circumstances. Model enforcers do need to support nesting (e.g. to determine whether to raise a warning when a request for an unknown resource is seen), but they will need to build the hierarchy of resources using the specified paths.
- Extra content after resource names is not allowed by default, but will be accepted if declared using the suffix attribute, as shown for **`/download.php`**.
- Values of the attributes name, alias and suffix are assumed to be patterns if they begin with a carat (**^**). They are treated as static text otherwise.

Profile Identification

In the second processing step we identify appropriate resource behavior. One resource can contain zero or many behaviors. Each behavior defines pre-conditions that must be fulfilled in order for the behavior to be selected for request processing.

In the example below, we document one resource with two behaviors: one for **`GET`** and another for **`POST`**.

```
<resource name="/sign-in.php">

   <behavior charset="utf-8">
      <preconditions>
         <match var="REQUEST_METHOD" value="GET" />
      </preconditions>
   </behavior>

   <behavior charset="utf-8">
      <preconditions>
         <match var="REQUEST_METHOD" value="POST />
      </preconditions>
   </behavior>

</resource>
```

Note:

- The above section allows for one resource to be invoked using either **GET**[7] or **POST**, but with no parameters.
- Each behavior can use its own charset, which will be used to correctly parse its parameters.

Secondary Parameter Extraction

Before parameters can be verified, we need to ensure we have a complete understanding of each behavior. While most parsing will take place in the initialization phase, we still need to take care of the parameters in non-standard places. We have to postpone dealing with such parameters until we have been able to identify the correct behavior, as every behavior can potentially use a different non-standard location for transport.

```
<resource name="/download.php" suffix="^/">
   <behavior>

      <preconditions>
         <match var="REQUEST_METHOD" value="GET" />
      </preconditions>

      <customParamsDefinition>
         <extractUsingPattern
               from="PATH_INFO"
               into="USER"
               pattern="^/(?P<filename>.+)$"
         />
      </customParamsDefinition>

      <params>

         <!--parameter definition
         omitted for clarity -->
      </params>

   </behavior>
</resource>
```

The example above extracts one parameter, named **filename**, out of variable **PATH_INFO** and into the group of parameters named **USER**. We are using regular expressions and named group capture to support extraction of more than one parameter and assign parameter names.

Although we suspected this mechanism of parameter extraction to be limited and unable to cope with every application, it works in most cases. Furthermore, the model can easily be extended to support other extraction mechanisms. The main drawback is that we cannot express arbitrary extraction logic in our XML format. Embedding script in the model would solve this problem, but would also significantly affect the portability of the solution.

[7] **HEAD** should also be allowed, in compliance with the HTTP protocol that requires **HEAD** to be treated as **GET**, but without the response body sent.

Parameter Verification

In the final and most interesting step, we look at the defined parameters for the identified behavior, and compare them to what was supplied in the request.

```
<params>
   <param name="m">
      <origins>
         <origin>QUERY_STRING</origin>
         <origin>REQUEST_BODY</origin>
      </origins>

      <!-- How many parameters are allowed? -->
      <cardinality min="1" max="3" />

      <!-- Length constraints. -->
      <length min="3" max="10" />

      <!-- Allowed byte values in content. -->
      <byteRanges>
         <range from="10" to="10" />
         <range from="13" to="13" />
         <range from="32" to="126" />
      </byteRanges>

      <!-- Content must match pattern. -->
      <content value="^\d+$" />
   </param>
</params>
```

Note:

- Notice how the validation is split into multiple steps, one for each parameter attribute. This allows us to easily extend the model to handle new attributes.
- We plan to introduce a data dictionary section where parameter types could be defined. For example, a **`userID`** parameter is likely to be used in many places in an application. Thus, there is no reason to force the model to declare it within every behavior. Instead, we allow such parameters to be defined only once and referred to using their aliases.

DEALING WITH UNCERTAINTY

Our examples so far have all assumed that we have a full understanding of the application we are modeling. But, as previously discussed, this will be true only in a limited number of usage scenarios. We propose to handle uncertainty with the addition of a *confidence* attribute to all model building blocks:

- Branching confidence.
  Are we confident that we have identified all resource branches (children)? This value can drive enforcement when an unidentified resource is observed.

- Resource confidence.
  Are we confident a resource correctly identifies all of its behaviors? This value can drive enforcement when a failure to identify a valid behavior occurs.
- Behavior confidence.
  Are we confident a behavior correctly identifies all of its parameters? This value can drive enforcement when an unidentified parameter is observed.
- Parameter confidence.
  Are we confident a parameters is accurately described by its attributes? This value can drive enforcement when a parameter fails validation.

The value of the confidence attribute itself is an integer between 0 and 100 (inclusive), the meaning of which is purposefully left undefined: we leave to the enforcers to choose how to interpret it. Having said that, assigning meanings to the extremes is easy:

- A confidence of 0 means that we have very little or no understanding of that part of the application and that no meaningful information can be extracted from the model.
- A confidence of 100 means that we believe that part of the model is complete and that it can be strictly enforced.

A confidence value from the remainder of the range will likely be used by the model enforcers to calculate the probability of the request being an attack. We believe most enforcers will also support multiple configurable settings for actions such as warning, blocking and other settings to explicitly what action should be taken for events such as unidentified resources, unidentified parameters and so on.

For manually produced partial models (e.g. virtual patches) we believe the best approach is to omit the confidence attribute, in which case the value will default to 100. The purpose of the confidence attribute is not to decide whether a request should be blocked or not; the user should always have this control.

LIMITATIONS

Our model, as currently defined, suffers from the following limitations:

- Limited content validation options.
  We only support content validation through regular expression patterns. Regular expressions are a very powerful, but they only allow for limited validation logic. An inclusion of validation language could make the implementation of any validation logic possible, with potentially only a small performance decrease.
- No support for content transformation.
  We can currently only analyze content as it appears. Reality has demonstrated that many applications perform further custom transformations, either on purpose or by mistake. Support for a pipeline of transformations along with a library of commonly-used transformation functions—similarly to what is available in ModSecurity—would fix this deficiency.
- No support for parameter hierarchies.
  Although our model supports groups of parameters that appear in different request locations, our groups are lists and there is no support for hierarchies. While this approach takes care of the way most

applications are built, it does not account for hierarchical data transport encodings, such as XML and JSON[13].

CONCLUSIONS AND FUTURE WORK

We have proposed to have applications stop unconditionally accepting every request, and instead serve only those requests that are known to be valid. By designing an abstract model based around HTTP, we ensure the concept can be deployed in many different scenarios, ranging from development to deployment.

We do not see this work as an end in itself. Rather, this paper should serve as an opening for a discussion among the interested parties: web application developers, web server developers, system administrators, computer security researchers and others. The concept presented here is yet to be proven in real life. Our next step is to test our ideas and tweak the model until it provides a reasonable success rate, defined as working for most web applications currently in production.

REFERENCES

[1] Ivan Ristic. Portable Web Application Firewall Rule Format. 2005.

[2] Marcus J. Ranum. What is "Deep Inspection"?

[3] David Scott and Richard Sharp. Abstracting Application-Level Web Security. Technical report, University of Cambridge, 2001.

[4] Jeff Williams *et al.* OWASP Stinger, 2003.

[5] Apache Struts, http://struts.apache.org.

[6] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS '03). 251-261 ACM Press. Washington, DC. October 2003.

[7] Ivan Ristic *et al*. ModSecurity (www.modsecurity.org), 2002.

[8] Christian Folini. REMO—Rule Editor for ModSecurity (remo.netnea.com), 2007.

[9] Christian Bockermann. Web Application Profiles (www.jwall.org), 2008.

[10] Philip Greenspun. Philip and Alex's Guide to Web Publishing. Morgan Kaufmann, 1999.

[11] Aspect Oriented Programming (AOP), http://en.wikipedia.org/wiki/Aspect-oriented_programming.

[12] Common Gateway Interface (CGI), 1995.

[13] Douglas Crockford. JavaScript Object Notation—JSON (www.json.org).