Enough With Default Allow in Web Applications!

Ivan Ristic, Ofer Shezaf
Breach Security (www.breach.com)

Revision 1 (June 30, 2008)

Abstract

The *default allow* deployment model, which is commonly used to implement and deploy web applications, is the cause of numerous security problems. We propose a method of modelling web applications in a platform-agnostic way to adopt a *default deny* model instead, removing several classes of vulnerability altogether and significantly reducing the attack surface of many others. Our approach is best adopted during development, but can be nearly as efficient as an afterthought, or when used at deployment time.

1 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 had happened with little thought about security. What was supposed to be a simple mechanism for document exchange exploded into an application delivery platform. Web-based application interfaces are now everywhere. So it's no doubt that we are finding ourselves wishing we could go back in time to do things right from the beginning. Instead, we are forced to waste enormous efforts on just making the present bearable. Our hands are now tied: improvements can only be made incrementally, as it is imperative to keep the Web running while we are improving it. 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 is an attempt at *both.*

Out of all time spent dealing with web application security, most is currently spent discovering problems, improving coding practices and applying band aids. We feel that, although these activities are unavoidable at this point, more time should be spent on changing the way applications are developed and deployed, so that classes of problem can be systematically eliminated. The only way to truly prevent security issues is to make sure they cannot be created in the first place. In other words, we must make it difficult—hopefully impossible—to shoot ourselves in the foot.

2 Goals

Our goal with this paper is to change one of the very important 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 fundamentally wrong, and completely the opposite of what we have to come to consider good in security.

Two current development practices are amplifying the problem further:

1. In developing web applications programmers are forced to interface directly with a number of different protocols and specifications—the main being HTTP—and all their complexities. We believe this is too much to ask, especially considering the current state of software development where focus is on delivering features with security as an afterthought. Furthermore, the situation with web security is so bad that even those who specialise in this subject are finding it difficult to cope with new developments. How can we expect the programmers to do a good job in such situation?

2. In many cases applications are built as collections of files, which web servers are instructed to process: scripts are executed, other files delivered verbatim. This practice makes applications mere shooting ducks, standing in the open with no protection whatsoever. Even a simple omission is likely to escalate into a vulnerability.

   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 but use a different extension. If a backup file makes its way to the web server, the web server is not going to know 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. Too much damage has already been done: given choice the majority would continue to develop in the way they are doing now. We feel that an alternative may be more feasible, especially one that can be used to put things right long term but also serve as an band-aid short term:

1. We are hoping that our proposal is on the right side of the balance of attractiveness and inconvenience to have a *chance* to be used.

2. Since our proposed solution can be applied externally, we are not appealing only to architects and developers, but to system administrators and security professionals

too. These groups are typically better motivated to address the security issues in the applications they handle.

2.1 Benefits

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

1. Information leakage, where data is leaked through files that are unintentionally distributed through web server.
2. Injection attacks through integer parameters can be prevented[1].
3. Injection attacks through parameters of other types can be significantly mitigated, depending on the parameter types.
4. Exploitation of so-called debug parameters, which were intended for debugging and troubleshooting, but were mistakenly left in production code.
5. Exploitation of any functionality other than production code, even if it was left in the application by mistake.
6. Buffer attacks are more difficult because parameter size limits can be enforced.
7. Attacks that exploit errors in web server configuration (e.g. attempts to use `PUT`, `DELETE`, or any of the WebDAV methods) can be eliminated.
8. The application attack surface can be reduced 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).

2.2 Use Cases

We have identified the following use cases:

Software developers Best results will be achieved if the default deny model is adopted in software development, but other options are possible:

[1] Susceptibility to injection attacks is a result of missing or inadequate encoding of data at system boundaries. Input validation, which is 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 any type of injection attacks when you are only able to use digits in attack payload.

1. New applications can be written with default deny in mind, ensuring the model and the application remain in sync, because developers will need to declare every external function they wish exposed.
2. Implementing default deny is going to be more difficult for existing applications because a good model will require a very good understanding of how an application works.
3. Even when default deny is not adopted in development, it can still be of significant use when it comes to fixing vulnerabilities quickly. Software developers can ease the pain experienced by its users by mitigating problems quickly by virtual patches, in parallel working on a proper fix.

Users Users are sometimes forced to live with insecure applications, for one reason or another. Faced with an application with a bad track record, or with an vendor that is slow to react, they can help themselves by building and deploying default deny models as own security shields[2]:

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 application source code.
2. Legacy applications are unlikely to get fixed, in many cases because no one understands how they work and dares to attempt a change in fear of breaking them. Security of such applications can be made bearable through a deployment of a default deny security model.
3. Users of popular applications could collaborate to build models together. In 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 to 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, especially those network-based, that have a good view of entire network segments. Such tools could be extended to support import of application models, either manually or programmatically. In the former case, administrators could feed application models to web application firewalls as part of deployment procedure. In the latter case, web vulnerability scanners could be configured to send virtual patches for every vulnerability they discover.

[2] We do not mean to say that every user should be a web application security specialist. Our point here is that users will be in control. Those who can do this job themselves probably will; other can hire security consultants to do the work for them. In either case, the user is in control of the situation, and that is our desire.

Web vulnerability scanners Web vulnerability scanners typically operate in two logical phases: crawling and testing. The crawling phase is very important: a missed resource will not be tested for security problems. Our proposal to document 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. The same effect can be achieved by having scanners communicate with web application firewalls on the same site, using the format proposed in this paper as the common language.

3 Previous Work

3.1 Default Deny

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

Ranum[2] has an insightful account of how the *default deny* culture lost to the *default allow* culture due to pressures for more 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 that 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—thus 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 limited in size but application often do not check whether the user provided data is within the limit. 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.

3.2 Related Work in Web Application Space

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

The OWASP Stinger project[6] is a centralised input validation component for Java that can be used with both new and existing applications (without the need to change application code, or have access to it), thanks to it being implemented as a Java Servlet Filter.

Kruegel and Vigna[7] 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[5] made designed a portable web application firewall rule format back in 2005 along with a Java-based implementation, but the idea failed to take off.

ModSecurity[4] is an open source web application firewall that can work either embedded or as a network gateway (coupled with an Apache reverse proxy). Its rule language is model agnostic (i.e. it supports the default allow and the default deny modes), but the lack of explicit support for positive security make its usage for anything other than small tasks (e.g. virtual patching) difficult. 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] wrote a tool that constructs a positive security model out of ModSecurity transaction logs (which contain full transaction content) and exports it back into ModSecurity rules. We are planning to use the same approach for our proof of concept.

4 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[3]. We basically treat HTTP as an API we can intercept, which is a view similar to that of AOP programming. This position allows us to ignore application implementation details, supporting any web application platform based on standards. 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.

[3] 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[1].

4.1 Requirements

In this section we describe the main requirements that guided us in designing the positive security model.

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 universally supported and easy to parse.

Partial model support To build a complete positive security model is only possible if the application is very simple or if the process is automated and incorporated into the development process.

- A person who sets out to build a model is likely to work by implementing model for one resource at a time.
- Due to time constraints, they may decide to work only on parts of the application or site they feel most exposed. (For example, the attack on the login page that is exposed on the public Internet is more likely to attract attacks rather 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 build the model for all resources in parallel (e.g as they are seeing the transactions), but they are not going likely to be able to build good model without seeing many transactions on 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.

Suitability for real-life Although much of the infrastructure used to develop web applications is standardised, that does not mean that application developers always choose to use it in a standardised 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 application, which are everything but written according to text-book examples.

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 they don't work as well with people.

4.2 Support For Non-Standard Behaviour

Parametarised URIs Many applications will not only transport parameters in the usual places (i.e. the query string and the request body), but embedded in the URL as well. This Amazon URL `http://www.amazon.com/dp/0596007248/`, for example, contains book ISDN number that a script on the server will extract from the URL and use to look the book up in the database.

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

Gateway pattern While in some application one request URI corresponds to one unit of work, many applications will perform further request routing internally using some request aspect. Such an application may have two, three or more (some applications are known to implement their entire functionality using a single gateway script) completely different request types processed by what—from the outside–appears to be a single resource.

The most common variations of the gateway design pattern are given below:

Request method Applications will often implement two behaviours 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 `POST` by processing the supplied `POST` data.

`PATH_INFO` `PATH_INFO` is the name of the variable in the CGI[10] specification that contains the part of the URL appended to the filename of the script. Applications often rely on the `PATH_INFO` information to implement external URLs that are user and search engine friendly. Unless `PATH_INFO` is parametarised, however, this case is automatically handled because, from the outside, each URL will have its own behaviour, 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*.

Dynamic 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.

4.3 Building Blocks

Application In many instances sites will contain one application, but this does not always has 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, which handles requests. In many cases it will be equivalent to a script, although there are many cases where this is not true.

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

Parameter In the end, applications must receive data and they do that through parameters. In our model each behaviour (function call) can receive zero or more named parameters.

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

5 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.

5.1 Initialisation

The main task of the initialisation 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`.

5.2 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 make a difference between folders and files, as such difference does not exist in the URL space. The following is an example of a simple web site containing a few scripts:

```
<applicationModel>

    <resource name="/" default="index.php">

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

        <resource name="sign-out.php" />

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

    </resource>

</applicationModel>
```

Note the following:

- URLs are relative to the root of the application, which may not necessarily be the same as the root of the 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.
- Path separators are not used anywhere in the model definition (excluding the use of the forward slash as the special name for the application root resource). Thus we leave the choice of path separators for deployment time.
- Resource names are case sensitive. Case-sensitivity is a deployment configuration option that is outside the scope of the application model.
- Note how one resource (`index.php`) was declared as the default one for the parent resource. This is to accommodate aliasing which is commonly used in all web servers.
- Extra content after resource names is not allow by default, but will be accepted if properly declared using the `suffix` attribute, as shown for `download.php`.
- Values of the attributes `name`, `default` and `suffix` will be assumed to be patterns if they begin with a carat (^). They will be treated as static text otherwise.

5.3 Profile Identification

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

In the example below we document on resource with two behaviours: one for `GET` and another for `POST`.

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

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

        <!-- remainder omitted for clarity -->
    </behaviour>

    <behaviour>
        <preconditions>
            <match var="REQUEST_METHOD" value="POST" />
        </preconditions>

        <!-- remainder omitted for clarity -->
    </behaviour>

</resource>
```

5.4 Secondary Parameter Extraction

Before parameters can be verified we need to ensure we have a complete understanding of the behaviour. While most parsing will take place in the initialisation 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 behaviour, as every behaviour can use a potentially different non-standard location for transport.

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

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

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

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

    </behaviour>
</resource>
```

The example above extract 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.

5.5 Parameter Verification

In the final and most interesting step we look at the defined parameters for the identified behaviour, 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>
```

6 Dealing With Uncertainty

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

Branching confidence Are we confident 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 its behaviours? This value can drive enforcement when a failure to identify a valid behaviour occurs.

Behaviour confidence Are we confident a behaviour correctly identifies all its parameters? This value can drive enforcement when an unidentified parameter is observed.

Parameter confidence Are we confident a parameter 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, 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:

1. A confidence of **0** means that we have very little or no understanding of that part of the application and that no meaningful information cannot be extracted from the model.

2. A confidence of **100** means that we believe that part of the model is complete and that it can be, for example, 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.

7 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 a limited validation logic. An inclusion of validation language could make the implementation of any validation logic possible, possibly on the account of a small performance decrease.

No support for content transformation We can currently only analyse content as it appears, but real life has demonstrated that many applications perform further custom transformations, either on purpose or by mistake. A 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[11].

8 Conclusions and Future Work

We have proposed to have applications stop accepting every request unconditionally, 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, we believe this paper, in its current form, should serve as an opening for a discussion among the interested parties: web application developers, web server developers, system administrators, computer security researchers, and so on. The concept presented here is yet to be proven in real life. Our next step is thus to test our ideas in real life and tweak the model until it provides a reasonable success rate, which we defined as working for most web applications currently out there.

References

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

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

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

[6] Jeff Williams et all. OWASP Stinger, 2003.

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

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

[9] Christian Bockermann. WebApplicationProfiler (www.jwall.org), 2008.

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

[11] Douglas Crockford. JavaScript Object Notation - JSON (www.json.org).