Use AWS WAF to Mitigate OWASP's Top 10 Web Application Vulnerabilities

July 2017

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

AWS WAF is a web application firewall that helps you protect your websites and web applications against various attack vectors at the HTTP protocol level. This paper outlines how you can use the service to mitigate the application vulnerabilities that are defined in the Open Web Application Security Project (OWASP) Top 10 list of most common categories of application security flaws. It's targeted at anyone who's tasked with protecting websites or applications, and maintaining their security posture and availability.

Introduction

The Open Web Application Security Project (OWASP) is an online community that creates freely available articles, methodologies, documentation, tools, and technologies in the field of web application security.[1] They publish a ranking of the 10 most-critical web application security flaws, which are known as the OWASP Top 10.[2] While the current version was published in 2013, a new 2017 Release Candidate version is currently available for public review.

The OWASP Top 10 represents a broad consensus of the most-critical web application security flaws. It's a widely accepted methodology for evaluating web application security and build mitigation strategies for websites and web-based applications. It outlines the top 10 areas where web applications are susceptible to attacks, and where common vulnerabilities are found in such workloads.

For any project aimed at enhancing the security profile of websites and web-based applications, it's a great idea to understand the OWASP Top 10 and how it relates to your own workloads. This will help you implement effective mitigation strategies.

AWS WAF is a web application firewall (WAF) you can use to help protect your web applications from common web exploits that can affect application availability, compromise security, or consume excessive resources.[3] With AWS WAF, you can allow or block requests to your web applications by defining customizable web security rules. Also, you can use AWS WAF to create rules to block common attack patterns, as well as specific attack patterns targeted at your application.

AWS WAF works with Amazon CloudFront,[4] our global content delivery network (CDN) service, and the Application Load Balancer option for Elastic Load Balancing.[5] By using these together, you can analyze incoming HTTP requests, apply a set of rules, and take actions based on the matching of those rules.

AWS WAF can help you mitigate the OWASP Top 10 and other web application security vulnerabilities because attempts to exploit them often have common

detectable patterns in the HTTP requests. You can write rules to match the patterns and block those requests from reaching your workloads.

However, it's important to understand that using any web application firewall doesn't fix the underlying flaws in your web application. It just provides an additional layer of defense, which reduces the risk of them being exploited. This is especially useful in a modern development environment where software evolves quickly.

Web Application Vulnerability Mitigation

In April 2017, OWASP released the new iteration of the Top 10 for public comment. The categories listed in the new proposed Top 10 are many of the same application flaw categories from the 2013 Top 10 and past versions:

A1	Injection
A2	Broken Authentication and Session Management
A3	Cross-Site Scripting (XSS)
A4	Broken Access Control (NEW)
A5	Security Misconfiguration
A6	Sensitive Data Exposure
A7	Insufficient Attack Protection (NEW)
A8	Cross-Site Request Forgery (CSRF)
A9	Using Components with Known Vulnerabilities
A10	Underprotected APIs (NEW)

The new A4 category consolidates the categories *Insecure Direct Object References* and *Missing Function Level Access Controls* from the 2013 Top 10. The previous A10 category *Unvalidated Redirects and Forwards* has been replaced with a new category that focuses on Application Programming Interface (API) security. In this paper, we discuss both old and new categories.

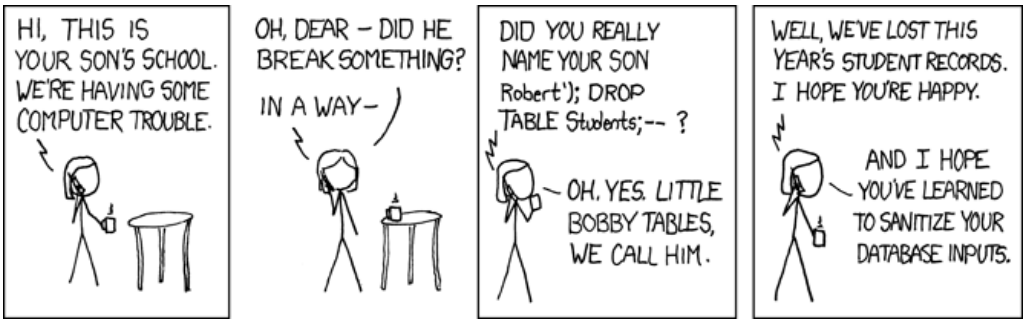
You can deploy AWS WAF to effectively mitigate a representative set of attack vectors in many of the categories above. It can also be effective in other categories. However, the effectiveness depends on the specific workload that's protected and the ability to derive recognizable HTTP request patterns. Given that the attacks and exploits evolve constantly, it's highly unlikely that any one web application firewall can mitigate all possible scenarios of an attack that targets flaws in any of these categories.

This paper describes recommendations for each category that you can implement easily to get started in mitigating application vulnerabilities. At the end of the paper, you can download an example AWS CloudFormation template that implements some of the generic mitigations discussed here. However, be aware that the applicability of these rules to your particular web application can vary.

A1 – Injection

Injection flaws occur when an application sends untrusted data to an interpreter.[6] Often, the interpreter has its own domain-specific language. By using that language and inserting unsanitized data into requests to the interpreter, an attacker can alter the intent of the requests and cause unexpected actions.

Perhaps the most well-known and widespread injection flaws are **SQL injection flaws**. These occur when input isn't properly sanitized and escaped, and the values are inserted in SQL statements directly. If the values themselves contain SQL syntax statements, the database query engine executes those as such. This triggers actions that weren't originally intended, with potentially dangerous consequences.



Credit: XKCD: Exploits of a Mom, published by permission.

Using AWS WAF to Mitigate

SQL injection attacks are relatively easy to detect in common scenarios. They're usually detected by identifying enough SQL reserved words in the HTTP request components to signal a potentially valid SQL query. However, more complex and dangerous variants can spread the malicious query (and associated key words) over multiple input parameter or request components, based on the

internal knowledge of how the application composes them in the backend. These can be more difficult to mitigate using a WAF alone—you might need to address them at the application level.

AWS WAF has built-in capabilities to match and mitigate SQL injection attacks. You can use a SQL injection match condition to deploy rules to mitigate such attacks.[7] The following table provides some common condition configurations:

HTTP Request Component to Match	Relevant Input Transformations to Apply	Justification
QUERY_STRING	URL_DECODE, HTML_ENTITY_DECODE	The **most common** component to match. Query string parameters are frequently used in database lookups.
URI	URL_DECODE, HTML_ENTITY_DECODE	If your application is using friendly, dirified, or clean URLs, then parameters might appear as part of the URL path segment—not the query string (they are later rewritten server side). For example: https://example.com/products/<product_id>/reviews/
BODY	URL_DECODE, HTML_ENTITY_DECODE	A **common** component to match if your application accepts form input. AWS WAF only evaluates the first 8 KB of the body content.
HEADER: Cookie	URL_DECODE, HTML_ENTITY_DECODE	A **less common** component to match. But, if your application uses cookie-based parameters in database lookups, consider matching on this component as well.
HEADER: Authorization	URL_DECODE, HTML_ENTITY_DECODE	A **less common** component to match. But, if your application uses the value of this header for database validation, consider matching on this component as well.

Additionally, consider any other components of custom request headers that your application uses as parameters for database lookups. You might want to match these components in your SQL injection match condition.

Other Considerations

Predictably, this detection pattern is less effective if your workload legitimately allows users to compose and submit SQL queries in their requests. For those cases, consider narrowly scoping an exception rule that bypasses the SQL injection rule for specific URL patterns that are known to accept such input. You can do that by using a **SQL injection match condition**, as described in the

preceding table, while listing the URLs that are excluded from checking by using a **string match condition**: [8]

```
Rule - action: BLOCK

when request matches SQL Injection Match Condition

and request does not match String Match Condition for excluded
Uniform Resource Identifiers (URI)
```

You can also mitigate other types of injection vulnerabilities against other domain-specific languages to varying degrees using string match conditions—by matching against known key words, assuming they're not also legitimate input values.

A2 – Broken Authentication and Session Management

Flaws in the implementation of authentication and session management mechanisms for web applications can lead to exposure of unwanted data, stolen credentials or sessions, and impersonation of legitimate users.[9] These flaws are difficult to mitigate using a WAF.

Broadly, attackers rely on vulnerabilities in the way client-server communication is implemented. Or they target how session or authorization tokens are generated, stored, transferred, reused, timed-out, or invalidated by your application to obtain these credentials. After they obtain credentials, attackers impersonate legitimate users and make requests to your web applications using those tokens.

For example, if an attacker obtains the JWT token that authorizes communication between your web client and the RESTful API, they can impersonate that user until the token expires by launching HTTP requests with the illicitly obtained authorization token. [10]

Using AWS WAF to Mitigate

Because illicit requests with stolen authorization credentials, sessions, or tokens are hard to distinguish from legitimate ones, AWS WAF takes on a reactive role.

After your own application security controls are able to detect that a token was stolen, you can add that token to a blacklist AWS WAF rule. This rule blocks further requests with those signatures, either permanently or until they expire. You can also automate this reaction to reduce mitigation time. AWS WAF offers an API to interact with the service.[11] For this kind of solution, you would use infrastructure-specific or application-specific monitoring and logging tools to look for patterns of compromise. Automation of AWS WAF rules is discussed in greater detail under A7 – Insufficient Attack Protection.

To build a blacklist, use a **string match condition**. The following table provides some example patterns:

HTTP Request Component to Match	Relevant Input Transformations to Apply	Relevant Positional Constraints	Values to Match Against
QUERY_STRING	Avoid exposing session tokens in the URI or QUERY_STRING because they're visible in the browser address bar or server logs and are easy to capture.		
URI			
HEADER: Cookie	URL_DECODE, HTML_ENTITY_DECODE	CONTAINS	Session ID or access tokens.
HEADER: Authorization	URL_DECODE, HTML_ENTITY_DECODE	CONTAINS	JWT token or other bearer authorization tokens.

You can use various mechanisms to help detect leaked or misused session tokens or authorization tokens. One mechanism is to keep track of client devices and the location where a user commonly accesses your application from. This gives you the ability to quickly detect if requests are made from an entirely different location or client device with the same tokens, and blacklist them for safety.

AWS WAF also supports rate-based rules. Rate-based rules trigger and block when the rate of requests from an IP address exceeds a customer-defined threshold (requests per 5-min interval). You can combine these rules with other predicates (conditions) that are available in AWS WAF. You can enforce rate-based limits to protect your applications' authentication or authorization URLs and endpoints against brute-force attack attempts to guess credentials. You can also use a string match condition to match authentication URI paths of the application:

HTTP Request Component to Match	Relevant Input Transformations to Apply	Relevant Positional Constraints	Values to Match Against
URI	URL_DECODE, HTML_ENTITY_DECODE	STARTS_WITH	/login *(or relevant application-specific URLs)*

This condition is then used inside a rate-based rule with the desired threshold for requests originating from a given IP address:

```
Rule - action: BLOCK; rate limit: 2000; rate key: IP
```

Only requests that match the string match condition are counted. When that count exceeds 2000 requests per 5-minute interval, the originating IP address is blocked. The minimum rate limit over a 5-minute you can set is 2000 requests.

A3 – Cross-Site Scripting (XSS)

Cross-site scripting (XSS) flaws occur when web applications include user-provided data in webpages that is sent to the browser without proper sanitization.[12] If the data isn't properly validated or escaped, an attacker can use those vectors to embed scripts, inline frames, or other objects into the rendered page (reflection). These in turn can be used for a variety of malicious purposes, including stealing user credentials by using key loggers, in order to install system malware. The impact of the attack is magnified if that user data persists server side in a data store and then delivered to a large set of other users.

Consider the example of a common, but popular, blog that accepts user comments. If user comments aren't correctly sanitized, a malicious user can embed a malicious script in the comments, such as:

```
 <script src="https://malicious-site.com/exploit.js"
type="text/javascript" />
```

The code then gets executed anytime a legitimate user loads that blog article.

Using AWS WAF to Mitigate

XSS attacks are relatively easy to mitigate in common scenarios because they require specific key HTML tag names in the HTTP request.

AWS WAF has built-in capabilities to match and mitigate XSS attacks. You can use a **cross-site scripting match condition** to deploy rules to mitigate these attacks.[13] The following table provides some common condition configurations:

HTTP Request Component to Match	Relevant Input Transformations to Apply	Justification
BODY	URL_DECODE, HTML_ENTITY_DECODE	A **very common component** to match if your application accepts form input. AWS WAF only evaluates the first 8 KB of the body content.
QUERY_STRING	URL_DECODE, HTML_ENTITY_DECODE	**Recommended** if query string parameters are reflected back into the webpage. An example is the current page number in a paginated list.
HEADER: Cookie	URL_DECODE, HTML_ENTITY_DECODE	**Recommended** if your application uses cookie-based parameters that are reflected back on the webpage. For example, the name of the user who is currently logged in is stored in a cookie and embedded in the page header.
URI	URL_DECODE, HTML_ENTITY_DECODE	**Less common**. But, if your application is using friendly, dirified URLs, then parameters might appear as part of the URL path segment, not the query string (they are later rewritten server side). There are similar concerns as with query strings.

Other Considerations

This detection pattern is less effective if your workload legitimately allows users to compose and submit rich HTML, such as the editor of a content management system (CMS).[14] For those cases, consider narrowly scoping an exception rule that bypasses the XSS rule for specific URL patterns that are known to accept such input, as long as there are alternate mechanisms to protect those excluded URLs.

Additionally, some image or custom data formats and match condition configurations can trigger elevated levels of false positives. Patterns that might indicate XSS attacks in HTML content can be legitimate in certain image or other data formats. For example, the SVG graphics format[15] also allows a `<script>` tag. You should narrowly tailor XSS rules to the type of request content that's expected if HTML requests include other data formats.

A4 – Broken Access Control

This category of application flaws, new in the proposed 2017 Top 10, covers lack of or improper enforcement of restrictions on what authenticated users are allowed to do. It consolidates the following categories from the 2013 Top 10: A4 – Insecure Direct Object References and A7 – Missing Function Level Access Controls.

Application flaws in this category allow internal web application objects to be manipulated without the requestor's access permissions being properly validated.[16] Depending on the specific workload, this can lead to exposure of unauthorized data, manipulation of internal web application state, path traversal, and file inclusion.

Your applications should properly check and restrict access to individual modules, components, or functions in accordance with the authorization and authentication scheme used by the application. Flaws in function-level access controls occur most commonly in applications where access controls weren't initially designed into the system, but were added later.[17]

These flaws also occur in applications that take a perimeter security approach to access validation. In these cases, access level can be validated once at the request initialization level. However, checks aren't done further in the execution cycle as various subroutines are invoked. This creates an implicit trust that the caller code can invoke other modules, components, or functions on behalf of the authorized user—which might not always hold true.

If your web application exposes different components to different users based on access level or subscription level, then you should have authorization checks performed anytime those functions are invoked.

Consider the following examples of flawed implementations for illustration:

1. A web application that allows authenticated users to edit their profile generates a link to the profile editor page upon successful authentication:

   ```
   https://example.com/edit/profile?user_id=3324
   ```

   The profile editor page, however, doesn't specifically check that the parameter matches the current user. This allows any user who's logged in to find information about any other user by simply iterating over the pool of user IDs. This exposes unauthorized information:

   ```
   https://example.com/edit/profile?user_id=3325
   ```

2. Another example is a helper server-side script that displays or allows a download of files for a document sharing site. It accepts the file name as a query string parameter:

   ```
   https://example.com/download.php?file=mydocument.pdf
   ```

   Somewhere in the script code, it passes the parameter to an internal file reading function:

   ```
   $content =
   file_get_contents("/documents/path/{$_GET[file]}");
   ```

   With no validation or sanitization and a vulnerable server configuration, the file parameter can be exploited to have the server read and reflect any file. For example:

   ```
   https://example.com/download.php?file=
   ..%2F..%2Fetc%2Fpasswd
   ```

   This is an example of both a **directory traversal** attack[18] and a **local file inclusion** attack.[19]

3. Consider a modular web application, which is a pattern popular with content management systems to enable extensibility, as well as applications using model-view-controller (MVC) frameworks. The entry point into the application is usually a router that invokes the right

controller, based on the request parameters after processing common routines (such as authentication/authorization):

```
https://example.com/?module=myprofile&view=display
```

A legitimate, authenticated user invoking the URL above should be able to see their own profile. A malicious user might authenticate and view their profile as well. However, they could attempt to alter the request URL and invoke an administrative module:

```
https://example.com/?module=usermanagement&view=display
```

If that particular module doesn't perform additional checks commensurate with the elevated privileges needed for administrators, it enables an attacker to gain access to unintended parts of the system.

Using AWS WAF to Mitigate

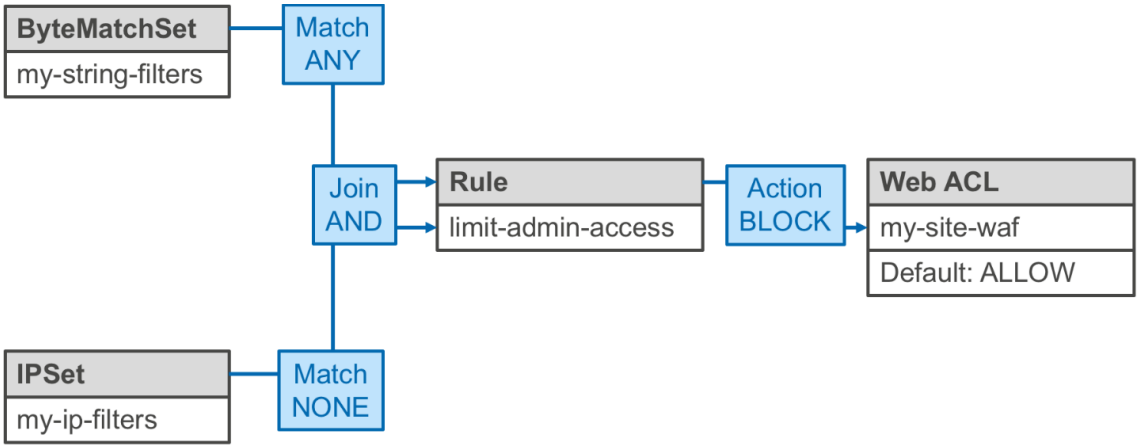
You can use AWS WAF to mitigate certain attack vectors in this category of vulnerabilities. Mitigating permission validation flaws is difficult using any WAF. This is because the criteria that differentiate good requests from bad requests are found in the context of the user (requestor) session and privileges, and rarely in the representation of the HTTP request itself. However, if malicious HTTP requests have a recognizable signature that legitimate requests don't have, you can write rules to match them.

Also, you can use AWS WAF to filter dangerous HTTP request patterns that can indicate path traversal attempts, or remote and local file inclusion (RFI/LFI). The table below illustrates a few such generic conditions:

HTTP Request Component to Match	Relevant Input Transformations to Apply	Relevant Positional Constraints	Values to Match Against
QUERY_STRING	URL_DECODE, HTML_ENTITY_DECODE	CONTAINS	../, ://
URI	URL_DECODE, HTML_ENTITY_DECODE	CONTAINS	../, ://

Also consider any other components of the HTTP request that your application uses to assemble or refer to file system paths. As with the patterns suggested in the previously discussed categories, these might be less effective if your application legitimately accepts URLs or complex file system paths.

If access to administrative modules, components, plugins, or functions is limited to a known set of privileged users, you can limit access to those functions by having them accessed from known source locations, a whitelisting pattern:



Other Considerations

If the authorization claims are transmitted from the client as part of the HTTP request and encapsulated using JWT tokens (or something similar), you can evaluate and compare them to the requested modules, plugins, components, or functions. Consider using AWS Lambda@Edge functions to prevalidate the HTTP requests and ensure that the relevant request parameters match the assertions and authorizations in the token.[20] You can use Lambda@Edge to reject nonconforming requests before they reach your backend servers.

A5 – Security Misconfiguration

Misconfiguration of server parameters, especially ones that have a security impact, can happen at any level of your application stack.[21] This can apply to the operating system, middleware, platform services, web server software, application code, or database layers of your application. Default configurations

that ship with these components might not always follow security recommendations or be fit for every workload.

A few examples of security misconfigurations are:

1. Leaving the Apache web server `ServerTokens Full` (default) configuration in a production system. This exposes the exact versions of the web server and associated modules in any server-generated responses. Attackers can use this information to identify known vulnerabilities in your server software.
2. Leaving default directory listings enabled on production web servers. This allows malicious users to browse for files that are hosted by the web server.
3. Application server configurations that return stack traces to end users on production systems in response to errors. Attackers can potentially discover the software components that are used. They might be capable of reverse engineering your code and potentially discovering flaws.
4. A previous feature in PHP. Several years ago, the default configuration for PHP allowed the registration of any request parameter (query string, cookie-based, POST-based) as a global variable. Since then, this feature has been deprecated and removed altogether. Coupled with a vulnerable version of PHP, it allowed for overwriting internal server variables via HTTP requests:

   ```
   http://example.com/?_SERVER[DOCUMENT_ROOT]=http://bad.com/bad.htm
   ```

   In a vulnerable application, this embeds a malicious site address in the site that users visit.

Using AWS WAF to Mitigate

You can use AWS WAF to mitigate attempts to exploit server misconfigurations in a variety of ways, as long as the HTTP request patterns that attempt to exploit them are recognizable. These patterns, however, are also application-stack specific. They depend on the operating system, web server, frameworks, or languages your code leverages. Generic rules that might not apply to your specific stack can be useful to you for nuisance protection because they block

requests that would otherwise be invalid, so your backend servers don't have to process them.

Here are a few strategies you can use:

- You should block access to the paths to administrative consoles, configuration, or status pages that are installed or enabled by default. Alternatively, you should restrict access to trusted source IP addresses, if they're in use. You should do this regardless of whether you specifically disabled or removed them (future actions might reactivate or reinstall them).
- Protect against known attack patterns that are specific to your platform, especially if you have legacy applications that rely on old platform behavior. For example, if you're using PHP, you might choose to block requests with a query string that contains `"_SERVER["`.

A whitelisting rule pattern, similar to the one discussed previously for the Broken Access Control category, can help with whitelisting specific subservices, such as the administrative console of a WordPress site.

Other Considerations

Also consider deploying Amazon Inspector to verify your software configurations.[22] It's an automated security assessment service that helps improve the security and compliance of applications that are deployed on AWS. Amazon Inspector automatically assesses applications for vulnerabilities or deviations from best practices.

To help you get started quickly, Amazon Inspector includes a knowledge base of hundreds of rules that are mapped to common security best practices and vulnerability definitions. Examples of built-in rules include checking for the enablement of the remote root login or the installation of vulnerable software versions. These rules are regularly updated by AWS security researchers.

In addition to detective controls, you can provide the best protection against attacks in this category by implementing and maintaining secure configurations. Configuration guidelines, such as the CIS Benchmarks[23], can help you deploy secure configurations. You can use services such as AWS Config[24] and Amazon

EC2 Systems Manager[25] to help you track and manage configuration changes over time.

A6 – Sensitive Data Exposure

Sensitive data exposure application flaws are typically harder to mitigate using web application firewalls.[26] These flaws commonly involve encryption processes that have been deficiently implemented. Some examples are the lack of encryption on transported or stored sensitive data, or using vulnerable legacy encryption ciphers,[27] where malicious parties can intercept and decode your data.

Less commonly, there can be flaws in application or protocol implementations, or client browsers, which can also lead to the exposure of sensitive data. Exploits that ultimately lead to sensitive data exposure can span multiple OWASP categories. A security misconfiguration that allows for the use of weak cryptographic algorithms leads to encryption downgrades and, ultimately, to an attacker being able to capture the data stream to decode sensitive data.

Using AWS WAF to Mitigate

Because the HTTP request is evaluated by AWS WAF after the incoming data stream has been decrypted, its rules have no impact on enforcing good encryption hygiene at the connection level.

Less commonly, if HTTP requests that can lead to sensitive data exposure have detectable patterns, you can mitigate them by using **string match conditions** that target those patterns. However, these patterns are application specific and require more in-depth knowledge of those applications.

For example, if your application relies heavily on the SHA-1 hashing algorithm,[28] malicious users might attempt to cause a hash collision using a pair of specially crafted PDF documents.[29] If your application allows uploads, it would be beneficial to set up a rule that blocks requests that contain portions of the base64-encoded representation of those files in the body.

When you attempt to block uploaded file signatures using AWS WAF, take into account the limits the service imposes on such rules. Uploaded data is base64

encoded. Therefore, your string match condition values have to be in base64 representation.

WAF searches the first 8 KB of the HTTP request body, or less if the multi-part encoding of the request body contains other field parameters that precede the file data itself. The relevant signature of the matched pattern can be up to 50 bytes in size. Most standardized file formats also have uniform preambles, so the first several bytes of the file are common to all files of that format. This forces you to derive the relevant signature from data further in the file.

Other Considerations

You can use other services in the AWS ecosystem to provide control over the encryption protocols and ciphers that are used at the connection level:

- For Elastic Load Balancing Classic Load Balancers,[30] you can select predefined or customized security policies.[31] These policies specify the protocols and ciphers that the load balancers can use to negotiate secure connections with clients.
- For Elastic Load Balancing Application Load Balancers,[32] you can select from a set of predefined security policies.[33] As with the Classic Load Balancers, these policies specify the allowed protocols and ciphers.
- For Amazon CloudFront,[34] our content delivery network (CDN) service, you can configure the minimum SSL protocol version you want to support,[35] as well as the SSL protocols you want CloudFront to use when it connects to your custom origins.

A7 – Insufficient Attack Protection

This category has been proposed for the new 2017 Top 10, and it reflects the reality that attack patterns can change quickly. Malicious actors are able to adapt their toolsets quickly to exploit new vulnerabilities and launch large-scale automated attacks to detect vulnerable systems. This category focuses strongly on your ability to react in a timely manner to new attack vectors and abnormal request patterns, or to application flaws that are discovered.

A broad range of attack vectors fall into this category, with many overlapping other categories. To better understand them, ask yourself the following questions:

- Can you enforce a certain level of hygiene at the request level? Are there HTTP request components that your application expects to exist, or can't operate without?
- Are you able to detect and recognize when your application is targeted with unusual request patterns or high volume? Do you have systems in place that can do that detection in an automated fashion? Are these systems capable of reacting to and blocking such unwanted traffic?
- Are you able to detect when a malicious actor launches a directed, targeted attack against your application, trying to find and exploit flaws in your application? Is this capability automated, so that you can react in near-real time?
- How fast can you deploy a patch to a discovered application flaw or vulnerability in your application stack and mitigate attacks against it? Do you have mechanisms in place to detect the effectiveness of the patch after deployment?

Using AWS WAF to Mitigate

You can use AWS WAF to enforce a level of hygiene for inbound HTTP requests. **Size constraint conditions**[36] help you build rules that ensure that components of HTTP requests fall within specifically defined ranges. You can use them to avoid processing abnormal requests. An example is to limit the size of URIs or query strings to values that make sense to the application. Also, you can use them to require the presence of specific headers, such as an API key for a RESTful API.

HTTP Request Component to Match	Relevant Input Transformations to Apply	Comparison Operator	Size
URI	NONE	GT (greater than)	*Maximum expected URI path size in bytes*
QUERY_STRING	NONE	GT	*Maximum expected size of the query string in bytes*
BODY	NONE	GT	*Maximum expected request body size in bytes*
HEADER:x-api-key	NONE	LT (less than)	1 *(or actual size of the API key)*
HEADER:cookie	NONE	GT	*Maximum expected cookie size in bytes*

You can use the example conditions described in this section with a blacklisting rule to reject requests that don't conform to the limits.

For detecting abnormal request patterns, you can use AWS WAF's rate-based rules that trigger when the rate of requests from an IP address exceeds your defined threshold (requests per 5-min interval). You can combine these rules with other predicates (conditions) that are available in AWS WAF.

For example, you can combine a rate-based rule with a string match rule to only count requests with a particular user-agent (say user-agent="abc"). This rule combination makes sure that only requests with user-agent="abc" are counted towards the determination of the rate violation by that IP address.

A key advantage of AWS WAF is its programmability. You can configure and modify AWS WAF web access control lists (ACLs), rules, and conditions by using a programmatic API at any time. Any changes normally take effect within a minute, even for our global service that's integrated with Amazon CloudFront.

Using the API, you can build automated processes that are able to react to application-specific abnormal conditions and take actions to block suspicious sources of traffic or notify operators for further investigation. These automations can operate in real-time, invoked via trap or honeypot URL paths. They can also be reactive, based on the analysis and correlation of application log files and request patterns.

As mentioned earlier, AWS provides a set of capabilities called the AWS WAF Security Automations.[37] These tools build upon the patterns highlighted previously. They use several other AWS services, most notably AWS Lambda for event-driven computing, and provide the following capabilities:[38]

- **Scanner and probe mitigation**. Malicious sources scan and probe internet-facing web applications for vulnerabilities. They send a series of requests that generate HTTP 4xx error codes. You can use this history to help identify and block IP addresses from malicious sources. This solution creates an AWS Lambda function that automatically parses access logs, counts the number of bad requests from unique source IP addresses, and updates AWS WAF to block further scans from those addresses.

- **Known attacker origin mitigation**. A number of organizations maintain reputation lists of IP addresses that are operated by known attackers, such as spammers, malware distributors, and botnets. This solution leverages the information in these reputation lists to help you block requests from malicious IP addresses.
- **Bots and scraper mitigation**. Operators of publicly accessible web applications have to trust that the clients accessing their content identify themselves accurately, and that they will use services as they're intended. However, some automated clients, such as content scrapers or bad bots, misrepresent themselves to bypass restrictions.

  This solution implements a honeypot that helps you identify and block bad bots and scrapers. In this solution, the honeypot URL is listed in the 'disallow' section of the robots.txt file.[39] Any IP that accesses this URL is, therefore, deemed malicious or noncompliant and is blacklisted.

Additionally, there are ways you might be able to use AWS WAF to mitigate newly discovered application flaws or vulnerabilities in your stack. They are discussed in greater detail later (see A9 – Using Components with Known Vulnerabilities).

A8 – Cross-Site Request Forgery (CSRF)

Cross-site request forgery attacks predominantly target state-changing functions in your web applications.[40] Consider any URL path and HTTP request that is intended to cause a state change (for example, form submission requests). Are there any mechanisms in place to ensure the user *intended* to take that action? Without such mechanisms, there isn't an effective way to determine whether the request is legitimate and wasn't forged by a malicious party. Depending solely on client-side attributes, such as session tokens or source IP addresses, isn't an effective strategy because malicious actors can manipulate and replicate these values.

CSRF attacks take advantage of the fact that all details of a particular action are predictable (form fields, query string parameters). Attacks are carried out in a way that takes advantage of other vulnerabilities, such as cross-site scripting or file inclusion—so users aren't aware that the malicious action is triggered using their credentials and active session.

Using AWS WAF to Mitigate

You can mitigate CSRF attacks by doing the following:

- Including unpredictable tokens in the HTTP request that triggers the action.
- Prompting users to authenticate for sending action requests.
- Introducing CAPTCHA challenges for sending action requests.[41]

The first option is transparent to end users—forms can include unique tokens as hidden form fields, custom headers, or, less desirably, query string parameters. The latter two options can introduce extra friction for end users and are generally only implemented for sensitive action requests. Additionally, CAPTCHAs can be circumvented by motivated actors and value combinations can also repeat.[42] As such, they are a less desirable mitigation control for CSRF.

You can use AWS WAF to check for the presence of those unique tokens. For example, if you decide to leverage a random universally unique identifier (UUIDv4)[43] as the CSRF token, and expect the value in a custom HTTP header named *x-csrf-token*, you can implement a **size constraint condition**:

HTTP Request Component to Match	Relevant Input Transformations to Apply	Comparison Operator	Size
HEADER:x-csrf-token	NONE	EQ (equal to)	36 *(bytes/ASCII characters, canonical format)*

You would build a blocking rule where requests **do not match** this condition (negated). You can further narrow the scope of the rule by only matching POST HTTP requests, for example. Build a rule using the negated condition above and an additional **string match condition** for:

HTTP Request Component to Match	Relevant Input Transformations to Apply	Relevant Positional Constraints	Values to Match Against
METHOD	LOWERCASE	EXACTLY	post

Other Considerations

Such rules are effective in filtering out CSRF attacks that circumvent your unique tokens. However, they aren't effective at validating if the request carries invalid, wrong, stale, or stolen tokens. This is because HTTP request introspection lacks access to your application context. Therefore, you need a server-side mechanism in your application to track the expected token or and ensure it's used exactly once.

As an example, the server sends a simple form to the client browser along with the embedded unique token as a hidden field. At the same time, it retains in the current server-side session store the token value it expects the browser to supply when the user submits the form. After the user submits the form, a POST request is made to the server that includes the unique hidden token. The server can safely discard any POST requests that don't contain the expected value for the supplied session. It should clear the value from the session store after it's used up, which ensures that the value doesn't get reused.

A9 – Using Components with Known Vulnerabilities

Currently, most web applications are highly composed. They use frameworks and libraries from a variety of sources, commercial or open source. One challenge is keeping up to date with the most recent versions of these components. This is further exacerbated when underlying libraries and frameworks use other components themselves.

Using components with known vulnerabilities is one of the most prevalent attack vectors.[44] They can help open up the attack surface of your web application to some of the other attack vectors discussed in this document. The decision to use such components can be an active trade-off to maintain compatibility with legacy code. Or, it's possible to inadvertently use vulnerable components if you're using components that depend on vulnerable subcomponents.

Mitigating vulnerabilities in such components is challenging because not all of them are reported and tracked by central clearinghouses such as Common Vulnerabilities and Exposures (CVE).[45] This puts the responsibility on the application developers to track the status of the components individually with the respective vendor, author, or provider. Often, vulnerabilities are addressed

in new versions of the components, including new enhancements, rather than fixing existing versions. This adds to the amount of work that developers have to perform to implement, test, and deploy the new versions of these components.

Using AWS WAF to Mitigate

The primary mechanism to mitigate known vulnerabilities in components is to have a comprehensive process in place that addresses the lifecycle of such components. You should have a way to identify and track the dependencies of your application and the dependencies of the underlying components. Also, you should have a monitoring process in place to track the security of these components.

Establish a software development process and policy that factors in the patch or release frequency of underlying components and acceptable licensing models. This can help you react quickly when component providers address vulnerabilities in their code.

Additionally, you can use AWS WAF to filter and block HTTP requests to functionality of such components that you aren't using in your applications. This helps reduce the attack surface of those components if vulnerabilities are discovered in functionality you're not using.

Does your application use server-side included components? These are usually files that contain code that is loaded at runtime to assemble the HTTP response directly or indirectly. Examples are Apache Server-side Includes[46] or code that loads via PHP *include*[47] or *require*[48] statements. Other languages and frameworks have similar constructs.

It's a best practice that these components aren't deployed in the public web path on your web server in the first place. However, sometimes this recommendation is ignored for a variety of reasons. If these components are present in the public web path, these files aren't designed to be accessed directly. Nevertheless, accessing them might expose internal application information or provide vectors of attack.

Consider using a **string match condition** to block access to such URL prefixes:

HTTP Request Component to Match	Relevant Input Transformations to Apply	Relevant Positional Constraints	Values to Match Against
URI	URL_DECODE	STARTS_WITH	/includes/ *(or relevant prefix in your application)*

Similarly, if your application uses third-party components but uses only a subset of the functionality, consider blocking exposed URL paths to functionality in those components that you don't use by using similar AWS WAF conditions.

Other Considerations

Penetration testing can also be an effective mechanism to discover vulnerabilities.[49] You can integrate it into your deployment and testing processes to both detect potential vulnerabilities, as well as to ensure that deployed patches correctly mitigate the targeted application flaws.

The AWS Marketplace[50] offers a wide range of vulnerability testing solutions from our partner vendors that are designed to help you get started easily and quickly. Keep in mind that AWS requires customers to obtain permission[51] before conducting such tests on resources that are hosted in AWS. However, some of the solutions available in the AWS Marketplace have been preauthorized, and you can skip the authorization step. They are marked as such in the solution title.

A10 – Underprotected APIs

Another new category proposed for the 2017 Top 10, Underprotected APIs, focuses on the target of potential attacks, rather than the specific application flaw patterns that can be exploited. This category recognizes the prevalence and anticipated future growth of APIs. Currently, entire applications are published that don't have a user-facing UI. Instead, they're available as APIs that other application publishers can use to build loosely coupled applications. Many applications can have both user UIs and APIs, whether those APIs are intended to be consumed by third parties or not.

The attack vectors are often the same as discussed in categories A1 through A9, and are common with more traditional web applications that are end user facing.

However, because APIs are designed for programmatic access, they do provide some additional challenges around security testing. It's easier to develop security test cases for user-facing UIs that have simpler data structures and more discrete, high-delay steps due to human interaction. In contrast, APIs are often designed to work with more complex data structures and use a wider range of request frequencies and input values. This is the case even if they're standardized and use well-known protocols, such as RESTful APIs[52] or SOAP.[53]

Using AWS WAF to Mitigate

Because the attack vectors for APIs are often the same as for traditional web applications, the mitigation mechanisms discussed throughout this document also apply to APIs in a similar manner. You can use AWS WAF in a variety of ways to mitigate these different attack vectors.

A key component that needs hardening is the protocol parser itself. With standardized protocols, it's relatively easy to extrapolate the parser used. With SOAP, you use XML[54]—and with RESTful APIs you will likely use JSON,[55] although you can also use XML, YAML,[56] or other formats. Thus, you can provide a critical success factor by effectively securing the configuration of the parser component and ensuring any vulnerabilities are mitigated.

As specific input patterns are discovered that would attempt to exploit flaws in the parser, you might be able to use AWS WAF **string match conditions** or **size restrictions** for the request body to block such request patterns.

Old Top 2013 A10 – Unvalidated Redirects and Forwards

Most websites and web applications contain mechanisms to redirect or forward users to other pages—internal or partner sites. If these mechanisms don't validate the redirect or forward requests,[57] it's possible for malicious parties to use your legitimate domain to direct users to unwanted destinations. These links use your legitimate and reputable domain to trick users.

Consider the following example:

> *You run a video sharing site and operate a URL shortener mechanism to enable users to share videos over text messages on mobile devices. You use a script to create the URLs:*
>
> `https://example.com/link?target=`
> **`https%3A%2F%2Fexample.com%2Fvideo%2Fe439853%3Fpos%3D200%26mode%3Dfullscreen`**
>
> *Users receive a URL like below, and it takes them to the correct content page:*
>
> `https://example.com/to?`**`vrejkR6T`**
>
> *If your link generator script doesn't validate the acceptable input domains for the target page, a malicious user can generate a link to an unwanted site:*
>
> `https://example.com/link?target=`
> **`https%3A%2F%2Fbadsite.com%2Fmalware`**
>
> *They can then package it and send it to users as it would originate from your site:*
>
> `https://example.com/to?`**`br09FtZ1`**

Using AWS WAF to Mitigate

The first step in mitigation is understanding where redirects and forwards occur in your application. Discovering what URL request patterns cause redirects, directly or indirectly, and under what conditions, helps you to build a list of potentially vulnerable areas. You should perform the same analysis for any exposed third-party components that your application uses in case they include redirect functionality.

If redirects and forwards are generated in response to HTTP requests from end users, as in the example above, then you can use AWS WAF to filter the requests and maintain a whitelist of domains that are trusted for redirect/forwarding

purposes. You can use a **string match condition** that targets the HTTP request component where the target parameter is expected to match a whitelist. In the example above, the set of conditions might look like the following:

1. Whitelist of allowed domains for redirects (block requests if no list value is matched):

HTTP Request Component to Match	Relevant Input Transformations to Apply	Relevant Positional Constraints	Values to Match Against
QUERY_STRING	URL_DECODE	CONTAINS	target=https://example.com
QUERY_STRING	URL_DECODE	CONTAINS	target=https://partnersite.com

2. Match only specific HTTP requests (to the redirector or router scripts):

HTTP Request Component to Match	Relevant Input Transformations to Apply	Relevant Positional Constraints	Values to Match Against
URI	URL_DECODE	STARTS_WITH	/link

You should combine these conditions in a single AWS WAF rule, which ensures that both conditions have to be met for requests to be matched.

Companion CloudFormation Template

We've prepared an AWS CloudFormation template[58] that contains a web ACL and the condition types and rules recommended in this document. You can use the template to provision these resources with just a few clicks (full API support is also available). Note that the template is designed as a starting point for you to build upon—and not as a production-ready, comprehensive set of rules. For more information about working with CloudFormation templates, see Learn Template Basics.[59]

The template is available at:

https://s3.us-east-2.amazonaws.com/awswaf-owasp/owasp_10_base.yml

The following example rules are included in the template:

- **Bad sources of traffic**. A generic IP block list rule that allows you to block requests from identified bad sources of traffic.
- **Broken access control:**
  - A path traversal and file injection rule that detects common file system path traversal, as well as local and remote file injection (LFI/RFI) patterns, to block suspicious requests.
  - A privileged module access restriction rule that limits access for administrative modules to known source IPs only. You can configure one path prefix and source IP address through the template. You can add additional patterns later by changing the conditions directly. For more information, see Creating and Configuring a Web Access Control List.[60]
- **Broken authentication and session management**. A block list that allows you to block illicit requests that use stolen or hijacked authorization credentials, such as JSON Web Tokens or session IDs.
- **Cross-site request forgery (CSRF)**. A rule that enforces the existence of CSRF-mitigating tokens.
- **Cross-site scripting (XSS)**. A rule that mitigates XSS attacks in common HTTP request components.
- **Injection**. A SQL injection rule that mitigates SQL injection attacks in common HTTP request components.
- **Insufficient attack protection**. A request-size hygiene rule that allows you to configure the maximum size of various HTTP request components by using template parameters, and block abnormal requests that exceed those maximum sizes.
- **Security misconfigurations**. A rule that detects some exploits of PHP-specific server misconfigurations. This rule might be less effective if you aren't running PHP-based applications, but it can still be valuable to filter out unwanted, automated HTTP requests that probe for PHP vulnerabilities.
- **The use of components with known vulnerabilities**. A rule that restricts access to publicly exposed URL paths that shouldn't be directly accessible, such as server-side include components or component features that aren't being used by your application.

We've chosen to package the example AWS WAF rule set as a CloudFormation template because it provides an easy and repeatable way to provision the whole rule set with a few simple clicks. The AWS CloudFormation documentation provides an easy-to-follow walkthrough about how to create a stack,[61] which is a collection of resources you can manage as a single unit.

Follow those instructions and provide the template on the **Select Template** page. Choose the option to **Upload a template to Amazon S3** and provide the downloaded template from your local computer. Otherwise, you can simply paste the template URL (https://s3.us-east-2.amazonaws.com/awswaf-owasp/owasp_10_base.yml) in the **Specify an Amazon S3 template URL** box.

On the **Specify Details** page, you can configure the template's parameters. A few key parameters to emphasize are:

- **Apply to WAF.** This parameter allows you to select whether you want to use the template to deploy a rule set for Amazon CloudFront web distributions or Application Load Balancers (ALB) in the current region. AWS WAF web ACLs get applied either to CloudFront web distributions or ALBs, depending on which service you use to deliver your application. The same stack can't be used for both, but you can deploy multiple stacks. You can also change this parameter's value later by updating the stack.
- **Rule effect.** This parameter determines the effect of your rule set. To minimize disruption, we recommend that you start with a rule set that **counts** matching requests. You can measure the effectiveness of your rules that way without impacting traffic. When you're confident about the effectiveness of your rules, you can deploy a stack that will **block** matching requests.

Continue following the AWS CloudFormation walkthrough instructions to deploy the stack. After you deploy the stack, you must associate the web ACL[62] that's deployed by the stack with your load balancer or web distribution resources to be able to use the rule set.

Conclusion

You can use AWS WAF to help you protect your websites and web applications against various attack vectors at the HTTP protocol level. As we discussed, in relation to OWASP security flaws, AWS WAF is very effective at mitigating vulnerabilities, to the extent that you can detect these attack patterns in HTTP requests.

Additionally, you can enhance the capabilities of AWS WAF with other AWS services to build comprehensive security automations. A set of such tools is available on our website in the form of the AWS WAF Security Automations.[63]

These tools enable you to build a set of protections that can react to the changing type of attacks your applications might be facing. The solution provides several easy-to-deploy automations in the form of a CloudFormation template for rate-based IP blacklisting, reputation list IP blacklisting, scanner and probe mitigation, bot and scraper detection, and blocking.

Contributors

The following individuals and organizations contributed to this document:

- Vlad Vlasceanu, Sr. Solutions Architect, Amazon Web Services
- Sundar Jayashekar, Sr. Product Manager, Amazon Web Services
- William Reid, Sr. Manager, Amazon Web Services
- Stephen Quigg, Solutions Architect, Amazon Web Services
- Matt Nowina, Solutions Architect, Amazon Web Services
- Matt Bretan, Sr. Consultant, Amazon Web Services
- Enrico Massi, Security Solutions Architect, Amazon Web Services
- Michael St.Onge, Cloud Security Architect, Amazon Web Services
- Leandro Bennaton, Security Solutions Architect, Amazon Web Services

Further Reading

For additional information, see the following:

- AWS WAF Security Automations: https://aws.amazon.com/answers/security/aws-waf-security-automations/
- OWASP Top 10 – 2017 rc1: https://github.com/OWASP/Top10/raw/master/2017/OWASP%20Top%2010%20-%202017%20RC1-English.pdf
- OWASP Top 10 – 2013: https://www.owasp.org/index.php/Top_10_2013

Document Revisions

Date	Description
July 2017	First publication

Notes

[1] https://www.owasp.org/

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

[3] https://aws.amazon.com/waf/

[4] https://aws.amazon.com/cloudfront/

[5] https://aws.amazon.com/elasticloadbalancing/applicationloadbalancer/

[6] https://www.owasp.org/index.php/Top_10_2013-A1-Injection

[7] http://docs.aws.amazon.com/waf/latest/developerguide/web-acl-sql-conditions.html

[8] http://docs.aws.amazon.com/waf/latest/developerguide/web-acl-string-conditions.html

[9] https://www.owasp.org/index.php/Top_10_2013-A2-Broken_Authentication_and_Session_Management

[10] https://jwt.io/

[11] http://docs.aws.amazon.com/waf/latest/APIReference/Welcome.html

[12] https://www.owasp.org/index.php/Top_10_2013-A3-Cross-Site_Scripting_(XSS)

[13] http://docs.aws.amazon.com/waf/latest/developerguide/web-acl-xss-conditions.html

[14] https://en.wikipedia.org/wiki/Content_management_system

[15] https://developer.mozilla.org/en-US/docs/Web/SVG

[16] https://www.owasp.org/index.php/Top_10_2013-A4-Insecure_Direct_Object_References

[17] https://www.owasp.org/index.php/Top_10_2013-A7-Missing_Function_Level_Access_Control

[18] https://en.wikipedia.org/wiki/Directory_traversal_attack

[19] https://en.wikipedia.org/wiki/File_inclusion_vulnerability

[20] http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html

[21] https://www.owasp.org/index.php/Top_10_2013-A5-Security_Misconfiguration

[22] https://aws.amazon.com/inspector/

[23] https://www.cisecurity.org/cis-benchmarks/

[24] https://aws.amazon.com/config/

[25] https://aws.amazon.com/ec2/systems-manager/

[26] https://www.owasp.org/index.php/Top_10_2013-A6-Sensitive_Data_Exposure

[27] https://en.wikipedia.org/wiki/Cipher

[28] https://en.wikipedia.org/wiki/SHA-1

[29] https://shattered.io/

[30] http://docs.aws.amazon.com/elasticloadbalancing/latest/classic/introduction.html

[31] http://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-ssl-security-policy.html

[32] http://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html

[33] http://docs.aws.amazon.com/elasticloadbalancing/latest/application/create-https-listener.html

[34] http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html

[35] http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/dis

tribution-web-values-specify.html#DownloadDistValuesMinimumSSLProtocolVersion

[36] http://docs.aws.amazon.com/waf/latest/developerguide/web-acl-size-conditions.html

[37] https://aws.amazon.com/answers/security/aws-waf-security-automations/

[38] https://aws.amazon.com/lambda/

[39] https://en.wikipedia.org/wiki/Robots_exclusion_standard

[40] https://www.owasp.org/index.php/Top_10_2013-A8-Cross-Site_Request_Forgery_(CSRF)

[41] https://en.wikipedia.org/wiki/CAPTCHA

[42] https://en.wikipedia.org/wiki/CAPTCHA#Circumvention

[43] https://en.wikipedia.org/wiki/Universally_unique_identifier

[44] https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities

[45] http://cve.mitre.org/

[46] https://httpd.apache.org/docs/current/howto/ssi.html

[47] http://php.net/manual/en/function.include.php

[48] http://php.net/manual/en/function.require.php

[49] https://en.wikipedia.org/wiki/Penetration_test

[50] https://aws.amazon.com/marketplace/search/results?x=0&y=0&searchTerms=vulnerability+scanner&page=1&ref_=nav_search_box

[51] https://aws.amazon.com/security/penetration-testing/

[52] https://en.wikipedia.org/wiki/Representational_state_transfer

[53] https://en.wikipedia.org/wiki/SOAP

[54] https://www.w3.org/XML/

[55] http://www.json.org/

[56] http://yaml.org/

[57] https://www.owasp.org/index.php/Top_10_2013-A10-Unvalidated_Redirects_and_Forwards

[58] https://aws.amazon.com/cloudformation/

[59] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/gettingstarted.templatebasics.html

[60] http://docs.aws.amazon.com/waf/latest/developerguide/web-acl.html

[61] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-create-stack.html

[62] http://docs.aws.amazon.com/waf/latest/developerguide/web-acl-working-with.html#web-acl-associating-cloudfront-distribution

[63] https://aws.amazon.com/answers/security/aws-waf-security-automations/