https://wiki.mozilla.org/WebAppSec/Web_Security_Verification

WebAppSec/Web Security Verification

< WebAppSec

Please also see the Secure Coding Guidelines document

Contents

Purpose

This document outlines the application security verification process. These actions are the basic verification steps that will generally apply to all applications. This is a base review standard and should be expanded and customized to the unique application.

Authentication

1. Does the application support logins?
    1. TLS - Verify the site is entirely HTTPS
    2. TLS - Verify that requests for HTTP URLs redirect to the equivalent HTTPS URL
    3. TLS - Verify that the Http Strict Transport Security flag is set
    4. Error Message - Verify that the error message displayed for an invalid username is the same message displayed for an invalid password
    5. Brute Force - Verify that a captcha is displayed after multiple (standard is 5) failed login attempts

2. Does the application manage its own set of user credentials?
    1. Verify that the application enforces the appropriate password complexity
    2. Verify a password blacklist is implemented
    3. If possible, verify that passwords are adequately protected in storage (bcrypt+hmac)
    4. If possible, verify that old password hashes are removed from the system
3. Does the application support administrative logins?
    1. Verify the admin login page is not publicly available
4. Does the application require an email verification before the account is activated?
    1. Verify that no actions can be taken or stored against the account until the verification link is followed
    2. Verify that the link only verifies ownership of the email account and does not log the user into the system
    3. Verify the code within the verification link is random
    4. Verify that the code within the verification link can not be used for an alternate user id or user account
    5. Verify that the verification code is invalidated after a single use
    6. Verify that the verification code is invalided after 8 hours if it is not used
5. Does the application provide a forgot password mechanism?
    1. Verify that no information is provided to indicate if a valid username or email address was entered
    2. Verify the code within the verification link is random
    3. Verify that the code within the verification link can not be used for an alternate user id or user account
    4. Verify that the verification code is invalidated after a single use
    5. Verify that the verification code is invalided after 24 hours if it is not used

Session Management

1. Does the application maintain state via a session identifier?
    1. Verify this session id is the default implementation and not a custom solution
    2. Verify the session identifier is 128-bit or larger
2. Does the application use cookies for the session identifier?
    1. Verify the SECURE flag is set for the cookies
    2. Verify the HTTPOnly flag is set for the cookies
    3. Verify the PATH and DOMAIN are appropriately set for the cookies
3. Does the application support logins?
    1. Verify a new session identifier is created for the user upon logging into the application
    2. Verify that upon logout the session id is expired on the client
    3. Verify that upon logout the session id is invalidated on the server
    4. Verify that critical applications enforce an inactivity timeout feature
4. Is session timeout appropriate for this application?
    1. Verify that authenticated sessions time out after a determined period of inactivity (15 minutes is recommended)

Access Control

1. Does the application support multiple users?
    1. Verify that two users within the same role cannot execute functionality on behalf of

the other account
    2. Verify that two users within the same role cannot access or edit data on behalf of the other account
2. Does the application support multiple roles?
    1. Verify that a user within one role cannot view features or functionality not available to the user
    2. Verify that a user within one role cannot execute functionality reserved for a different role
    3. Verify that a user within one role cannot access or edit data reserved for a different role

Input Validation

1. Does the application accept user input?
    1. Verify a sampling of input locations to ensure that reasonable maximums are in place when accepting user data
    2. Verify a sampling of input locations to ensure that the application allows only a defined set of acceptable characters
    3. Verify a sampling of input locations to ensure that the application is not relying solely on JavaScript validation
2. Does the application accept user input that is later viewable or returned to the user within a HTTP response?
    1. Verify a sampling of input and output locations to ensure user supplied content is properly encoded in the response
3. Does the application accept rich user content?
    1. Is HTML Purifier / AntiSamy / Bleach / Equivalent being used?
4. Input Encoding
    1. Verify that the application rejects invalid encoding (e.g. UTF-7 encoded content submitted in a UTF-8 request)
    2. Verify that the application rejects double encoded data, or properly decodes user input before applying validation or sanitization code

Output Encoding

1. Does the application send any untrusted data to the user's browser?
    1. Verify that robust input validation is in place
    2. Verify that all user controlled data is appropriately encoded; this is context dependent:
        1. HTML element content - the data must be HTML encoded
        2. HTML attributes - the data must be HTML attribute encoded
        3. HTML style attributes:
            - data must only be placed into property values
            - the value must not contain 'expression'
            - the value must be CSS escaped
        4. URLs - the scheme must be whitelisted, the URL must be URL escaped
        5. Script - data must only be placed in quoted values and must be script encoded
    3. Verify that all escaping and encoding routines escape / encode by default and have a whitelist for 'allowed' characters
2. Does the application send any untrusted data to a SQL database?
    1. Verify robust input validation is in place
    2. Verify that, if available on the target platform, only parameterized SQL queries are

used:
        1. Should parameterized queries not be available, ensure that database platform specific escaping is performed on all untrusted data.
3. Does the application invoke any OS commands?
    1. Verify that, where possible, user input is not sent to the OS at all
    2. If user data is sent to the OS:
        1. Verify that robust input validation is in place
        2. Verify that a robust escaping routine is in place to ensure special characters don't get to the command interpreter. Any escaping routine will need to be OS specific.

Cross-Domain

1. Does every state change require a CSRF Token?
    1. Verify that the same request without token will not succeed
    2. Verify that changing the HTTP method verb will be rejected
    3. Verify that CSRF tokens are
        - unique per user
        - random
        - contain enough entropy to be unpredictable
        - in a hidden field for form-elements
2. Cross-Framing / Clickjacking
    1. Verify that X-Frame-Options is set to DENY or SAMEDOMAIN
3. Third-Pary Scripts
    1. Verify that JavaScript from third party is locally hosted
    2. Verify that this script is reviewed
4. Third-Party Code Snippets (Tweet this, Like This etc.)
    1. Verify that the whole communication is using HTTPS
    2. Verify that authenticated content is only presented using HTTPS
    3. Verify that simply loading said snippets does not perform requests to the third party (Privacy)
    4. OAUTH
        - Verify that the initial form and further traffic is sent only via HTTPS
5. For Cross Origin Resource Sharing (CORS) see Cross Origin Resource Sharing & Access-Control headers

Secure Transmission

1. Does the web application support authenticated sessions?
    1. Verify all points from the login page to logout are served over HTTPS
    2. If the application handles its own logins:
        - Verify that any login pages are served over HTTPS
        - Verify that any login pages POST login data HTTPS
    3. Verify that all resources for HTTPS pages are also served over HTTPS
    4. Verify for all pages served over HTTPS (detailed above) that the resource is not also accessible over HTTP
    5. Verify the application uses STS headers
2. Does the application use a thick client for access instead of a client browser?
    1. Verify that invalid certificates are appropriately rejected

    2. Verify that user supplied credentials are not insecurely stored on the client device.
3. Is SSL deployed correctly?
    1. Verify the following, see https://www.owasp.org/index.php/Testing_for_SSL-TLS_(OWASP-CM-001)
        - SSLv3 or TLS only - no SSLv2
        - Check for SSL renegotiation bug
        - Verify that no connections <128 bit are allowed
        - Verify that no Export-strength algorithms are enabled on the server

Content-Security Policy

1. If the website is already using CSP
    1. Verify that the policy is appropriately scoped and restrictive
        - Verify that the policy is present and working
        - Verify that not everything is whitelisted
        - Verify that CSP violations will lead to a report in staging/development mode and that these reports will be reviewed
        - Verfiy that CSP violations will lead to a block in production mode
        - Verfiy that the whitelisted sites for static content (like JavaScript, CSS, Images, Fonts) do not generate dynamic content, which might be subject to misuse
2. If the website is not, but plans to do so
    1. Verify that contents like CSS and JavaScript are not inline
    2. Verify that CSS, JavaScript, Images, Fonts are hosted on a specific domain which gives out static content only

Logging

1. General Logging
    1. Verify that a standard logging format or library is used
    2. Verify that logging levels are configurable
    3. Verify that log entries have a reasonable constraint on size
    4. Verify that log storage paths are not accessible to the public
2. Log Injection
    1. Verify that no string formatting functions are invoked after integration of user controlled data
    2. Verify that user controlled data is sanitized or passed through an explicit whitelist
    3. Verify that binary data is encoded in a text format such as Base64
3. Verify that these events are logged
    1. Access Denied
    2. Admin Account Password Reset Request
    3. Admin Account Password change
    4. New admin account created

Admin login pages

1. Bruteforcing the authentication parameters is blocked by one of the following methods
    1. Admin login only accessible via SSL-VPN
    2. Account lockout

    3. acccess to admin login restricted to a whitelisted IP
    4. verify that 5 failed logins requires the user to solve a captcha
2. Verify that web pages for login and admin are accessible only via HTTPS
3. Verify that the Session cookie set to httponly and secure

Uploads

1. General uploads
    1. Verify that filenames are generated randomly via a process that will not result in path control characters, and do not contain any user input
    2. Make sure all OS calls are made in a command-safe way, such as the following:
        - Avoids the use of the interpreter entirely, or provides a parameterized interface
        - Verify that APIs which appear parameterized don't allow injection under the hood
        - White-list input
        - Least preferable - escape input. Ref implementation: http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/Encoder.html
    3. Verify maximum file size is limited
    4. Verify that only the expected file types are working
        - Verify that only a white list of known extensions are accepted
        - Verify that a library such as libmagic is used to analyze the contents of files to match binary file formats with approved file formats
    5. Verify that the files are served from a different domain
    6. Verify that platform sensitive formats (such as Python pickle) are not processed using unsafe API that could result in the file being processed
    7. Verify that files uploaded cannot be invoked as scripts on the server (e.g. python, php, javascript, etc)
2. Verify that images are
    1. Read
    2. Validated
        - Test with oversized images, overly large dimensions, or bogus files
        - Filename/minetype should be tested to verify matching file content
    3. Stripped of metadata (i.e. EXIF, XMP, IIM, etc - see http://en.wikipedia.org/wiki/Metadata#Photographs)
    4. Written back to disk, as described in "general uploads"
3. Archives
    1. Verify that decompressed maximum size is limited
    2. Verify that file extension and detected filetype match
    3. For structural requirements (e.g. AddOns): Verify file hierarchy
    4. Verify that password protected or encrypted archives are rejected
    5. Verify that nested archives are rejected

Error handling

1. Verify that error pages (e.g. 404 and 500) do not contain sensitive information (path, system status or setup information, debug messages)
2. Verify that global debug mode is disabled
3. Verify that user input in error messages is correctly encoded (XSS) and newlines are not written to text files as is.

Encryption

1. Add checklists for
    1. Encrypted tokens
        - ecb mode attacks, such as http://blog.portswigger.net/2011/10/breaking-encrypted-data-using-burp.html
        - cbc mode attacks, such as http://portswigger.net/burp/help/intruder.html#sourcebitflipper
    2. symmetric
        - algorithm
        - implementation
    3. asymmetric
        - algorithm
        - implementation
    4. hashing/hmac
    5. key storage

Existing OWASP checkist: https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet

Navigation menu

How to Contribute

- All-hands meeting
- Other meetings
- Contribute to Mozilla
- Mozilla Reps
- Student Ambassadors

MozillaWiki

Around Mozilla

Tools

WebAppSec/Secure Coding Guidelines

< WebAppSec

Contents

Introduction

The purpose of this page is to establish a concise and consistent approach to secure application development of Mozilla web applications and web services. The information provided here will be focused towards web based applications; however, the concepts can be universally applied to applications to implement sound security controls and design.

This page will largely focus on secure guidelines and may provide example code at a later time.

Status

The secure coding guidelines page is a living document and constantly updated to reflect new recommendations and techniques. Information that is listed is accurate and can be immediately used to bolster security in your application. If you have comments, suggestions or concerns please email mcoates <at> mozilla.com

Layout

The guidelines are discussed within logical security areas. Instead of discussing how to prevent each and every type of attack, we focus on a secure approach to designing an application. Within each section there is a listing of the types of the attacks these controls are geared to protect against. However, this document is not intended to serve as an in-depth analysis of the attack types, rather a guide to creating a secure application.

Easy Quick Wins

Here are a few items that are often missed and are relevant for most every website.

- For all cookies set the HTTPOnly and Secure flag
- Make sure login pages are only served on HTTPS and all authenticated pages are only served on HTTPS
- Don't trust any user data (input, headers, cookies etc). Make sure to validate it before using it

Secure Coding Guidelines

Authentication

Attacks of Concern

- online & offline brute force password guessing
- user enumeration
- mass account lockout (Account DoS)
- offline hash cracking (time trade-off)
- lost passwords

Password Complexity

All sites should have the following base password policy:

- Passwords must be 8 characters or greater
- Passwords must require letters and numbers
- Blacklisted passwords should be implemented (contact infrasec for the list)

Critical Sites

Examples: addons.mozilla.org, bugzilla.mozilla.org, or other critical sites.

Critical sites should add the following requirements to the password policy:

- Besides the base policy, passwords should also require at least one or more special

characters.

Password Rotation

Password rotations have proven to be a little tricky and this should only be used if there is lack of monitoring within the applications and there is a mitigating reason to use rotations. Reasons being short password, or lack of password controls.

- Privileged accounts - Password for privileged accounts should be rotated every: 90 to 120 days.
- General User Account - It is also recommended to implement password rotations for general users if possible.
- Log Entry - an application log entry for this event should be generated.

Account Lockout and Failed Login

Account Lockouts vs login failures should be evaluated based on the application. In either case, the application should be able to determine if the password being used is the same one over and over, or a different password being used which would indicate an attack.

The error message for both cases should be generic such as:

Invalid login attempts (for any reason) should return the generic error message

```
The username or password you entered is not valid
```

Logging will be critical for these events as they will feed up into our security event system and we can then take action based on these events. The application should also take action. Example would be in the case that the user is being attacked, the application should stop and/or slow down that user progress by either presenting a captcha or by doing a time delay for that IP address. Captcha's should be used in all cases when a limit of failed attempts has been reached.

Password Reset Functions

The password reset page will accept the username and then send an email with a password reset link to the stored email address for that account.

The following message should be returned to the user regardless if the username or email address is valid:

```
An email has been sent to the requested account with further information. If
you do not receive an email then please confirm you have entered the same
email address used during account registration.
```

We do not want to provide any information that would allow an attacker to determine if an entered username/email address is valid or invalid. Otherwise an attacker could enumerate valid accounts for phishing attacks or brute force attack.

Email Change and Verification Functions

Email verification links should not provide the user with an authenticated session.

Email verification codes must expire after the first use or expire after 8 hours if not used.

Password Storage

Separate from the password policy, we should have the following standards when it comes to storing passwords:

- Passwords stored in a database should using the hmac+bcrypt function.

The purpose of hmac and bcrypt storage is as follows:

- bcrypt provides a hashing mechanism which can be configured to consume sufficient time to prevent brute forcing of hash values even with many computers
- bcrypt can be easily adjusted at any time to increase the amount of work and thus provide protection against more powerful systems
- The nonce for the hmac value is designed to be stored on the file system and not in the databases storing the password hashes. In the event of a compromise of hash values due to SQL injection, the nonce will still be an unknown value since it would not be compromised from the file system. This significantly increases the complexity of brute forcing the compromised hashes considering both bcrypt and a large unknown nonce value
- The hmac operation is simply used as a secondary defense in the event there is a design weakness with bcrypt that could leak information about the password or aid an attacker

A sample of this code is here: https://github.com/fwenzel/django-sha2

Keep in mind that while bcrypt is secure you should still enforce good passwords. As slow as an algorithm may be if a password is "123" it still would only take a short amount of time before somebody figures it out.

Old Password Hashes

- Password hashes older than a year should be deleted from the system.
- After a password hash migration, old hashes should be removed within 3 months if user has yet to log in for the conversion process.

Migration

The following process can be used to migrate an application that is using a different hashing algorithm than the standard hash listed above. The benefits of this approach is that it instantly upgrades all hashes to the strong, recommended hashing algorithm and it does not require users to reset their passwords.

Migration Process
Migrate all password hashes entries in the database as follows. This is a one time, offline migration.

Stored in databases in form: {algo}${salt}${migration_hash}

```
* {algo} is {sha512+MD5},
* {salt} is a salt unique per-user,
* {migration_hash} is SHA512(salt + existingPasswordHash)
```

New hash process for new accounts or password changes
Use standard hashing process [above]

New Login Process
1. Attempt to login user with migration hash. This involves performing the old password hash procedure then adding the salt and finally performing the sha512.

```
Example: Old password hash process is md5
Migration Hash = sha512(perUserSalt + md5(user supplied password))
```

2. If authentication via migration hash is successful:

- Use the user's provided password and calculate the New Hash per the algorithm defined above.

- Overwrite the Migration Hash with the New Hash

3. If authentication via migration hash is NOT successful:

- The user may already be on the New Hash. Attempt to directly authenticate using the new hash. If this fails, then the password provided by the user is wrong.

Session Management

Attacks of Concern: Session Hijacking, Session Fixation, Brute Forcing Valid Session IDs

Session ID Length

Session tokens should be 128-bit or greater

Session ID Creation

The session tokens should be handled by the web server if possible or generated via a cryptographically secure random number generator.

Inactivity Time Out

Authenticated sessions should timeout after determined period of inactivity - 15 minutes is recommended.

Secure Flag

The "Secure" flag should be set during every set-cookie. This will instruct the browser to never send the cookie over HTTP. The purpose of this flag is to prevent the accidental exposure of a cookie value if a user follows an HTTP link.

Code Examples

HTTP-Only Flag

The "HTTP-Only" flag should be set to disable malicious script access to the session ID (e.g. XSS)

Login

New session IDs should be created on login (to prevent session fixation via XSS on sibling domains or subdomains).

Logout

Upon logout the session ID should be invalidated on the server side and deleted on the client via expiration/overwriting the value.

Access Control

Attacks of Concern Enumeration of site features for targeted attacks, Execution of unauthorized functionality, View or modify unauthorized data

Presentation Layer

Display Features and Functions Granted to User

It is recommended to not display links or functionality that is not accessible to a user. The purpose is to minimize unnecessary access controls messages and minimize privileged information from being unnecessarily provided to users.

Business Layer

Check Access Control Before Performing Action

Ensure that an access control check is performed before an action is executed within the system. A user could craft a custom GET or POST message to attempt to execute unauthorized functionality.

Data Layer

Check Access Control with Consideration of Targeted Data

Ensure that an access control check also verifies that the user is authorized to act upon the target data. Do not assume that a user authorized to perform action X is able to necessarily perform this action on all data sets.

Input Validation

Attacks of Concern: Introduction of Dirty/Malformed Data

Goal of Input Validation

Input validation is performed to minimize malformed data from entering the system. Input Validation is NOT the primary method of preventing XSS, SQL Injection. These are covered in output encoding below.

Input Validation Must Be:

- Applied to all user controlled data
- Define the types of characters that can be accepted (often U+0020 to U+007E, though most special characters could be removed and control characters are almost never needed)
- Defines a minimum and maximum length for the data (e.g. {1,25})

Examples of Good Input Validation Approaches For each field define the types of acceptable characters and an acceptable number of characters for the input

- Username: Letters, numbers, certain special characters, 3 to 10 characters
- Firstname: Letters, single apostrophe, dash, 1 to 30 characters
- Simple US Zipcode: Numbers, 5 characters

Note: These are just examples to illustrate the idea of whitelist input validation. You'll need to adjust based on the type of input you expect.

JavaScript vs Server Side Validation

Be aware that any JavaScript input validation can be bypassed by an attacker that disables JavaScript or uses a Web Proxy. Ensure that any input validation performed by JavaScript is also performed server side as well.

Positive Approach

The variations of attacks are enormous. Use regular expressions to define what is good and then deny the input if anything else is received. In other words, we want to use the approach "Accept Known Good" instead of "Reject Known Bad"

```
Example A field accepts a username. A good regex would be to verify
that the data consists of the following [0-9a-zA-Z]{3,10}. The data
is rejected if it doesn't match.

A bad approach would be to build a list of malicious strings and then
just verify that the username does not contain the bad string. This
approach begs the question, did you think of all possible bad strings?
```

Robust Use of Input Validation

All data received from the user should be treated as malicious and verified before using within the application. This includes the following

- Form data
- URL parameters
- Hidden fields
- Cookie data
- HTTP Headers
- Essentially anything in the HTTP request

Validating Rich User Content

It is very difficult to validate rich content submitted by a user. Consider more formal approaches such as HTML Purifier (PHP) or AntiSamy or bleach (Python)

Output Encoding

Output encoding is the primary method of preventing XSS and injection attacks. Input validation helps minimize the introduction of malformed data, but it is a secondary control.

Attacks of Concern: Cross Site Scripting, SQL/OS/LDAP/XML Injection

Preventing XSS

- All user data controlled must be encoded when returned in the html page to prevent the execution of malicious data (e.g. XSS). For example <script> would be returned as <script>
- The type of encoding is specific to the context of the page where the user controlled data is inserted. For example, HTML entity encoding is appropriate for data placed into the HTML body. However, user data placed into a script would need JavaScript specific output encoding

Detailed information on XSS prevention here: OWASP XSS Prevention Cheat Sheet

Preventing SQL Injection

- String concatenation to build any part of a SQL statement with user controlled data creates a SQL injection vulnerability.

- Parameterized queries are a guaranteed approach to prevent SQL injection.
- It's not realistic to always know if a piece of data is user controlled, therefore parameterized queries should be used whenever a method/function accepts data and uses this data as part of the SQL statement.

Further Reading: SQL Injection Prevention Cheat Sheet

Preventing OS Injection

- Avoid sending user controlled data to the OS as much as possible
- Ensure that a robust escaping routine is in place to prevent the user from adding additional characters that can be executed by the OS (e.g. user appends | to the malicious data and then executes another OS command). Remember to use a positive approach when constructing escaping routinges. Example

Further Reading: Reviewing Code for OS Injection

Preventing XML Injection

- Same approach as OS injection. In addition to the existing input validation, define a positive approach which escapes/encodes characters that can be interpreted as xml. At a minimum this includes the following: < > " ' &
- If accepting raw XML then more robust validation is necessary. This can be complex. Please contact the infrastructure security team for additional discussion

Cross Domain

Attacks of Concern: Cross Site Request Forgery (CSRF), Malicious Framing (Clickjacking), 3rd Party Scripts, Insecure Interaction with 3rd party sites

Preventing CSRF

An attacker creates a self posting form or image tag which executes an action on behalf of the authenticated user. Read more about this attack type here

- Any state changing operation requires a secure random token (e.g CSRF token) to prevent against CSRF attacks
- Characteristics of a CSRF Token
    - Unique per user & per user session
    - Tied to a single user session
    - Large random value
    - Generated by a cryptographically secure random number generator
- The CSRF token is added as a hidden field for forms or within the URL if the state changing operation occurs via a GET
- The server rejects the requested action if the CSRF token fails validation

Note: Some frameworks (such as django) provide this capability. Use the established CSRF protection from the framework instead of creating your own.

Preventing Malicious Site Framing (ClickJacking)

A newer attack that uses page layering and framing to convince the user to click or enter data on particular parts of the screen. These actions are actually sent to the framed site to perform actions

unbeknown to the victim user. Read more about this attack type here

Set the x-frame-options header for all responses containing HTML content. The possible values are "DENY" or "SAMEORIGIN".

- DENY will block any site (regardless of domain) from framing the content.
- SAMEORIGIN will block all sites from framing the content, except sites within

the same domain.

The "DENY" setting is recommended unless a specific need has been identified for framing.

Code Examples

3rd Party Scripts

- Careful consideration should be used when using third party scripts. While I am sure everybody would do an initial review, updates to scripts should be reviewed with the same due diligence.
- Ensure any scripts that are used are hosted locally and not dynamically referenced from a third party site.

Connecting with Twitter, Facebook, etc

- If using OAuth make sure the entire chain of communication is over HTTPS. This includes the initial OAuth request and any URLs passed as parameters.
- If redirecting to a login page for the app itself, ensure that URL is HTTPS and also that the selected URL does not simply redirect to a HTTP version
- Ensure the "tweet this" or "like this" button does not generate a request to the 3rd party site simply by loading the Mozilla webpage the button is on (e.g. no requests to third party site without user's intent via clicking on the button)

Secure Transmission

Attacks of Concern: Man in the middle, password theft, session id theft

When To Use SSL/TLS

- All points from the login page to the logout page must be served over HTTPS.
- Ensure that the page where a user completes the login form is accessed over HTTPS. This is in addition to POST'ing the form over HTTPS.
- All authenticated pages must be served over HTTPS. This includes css, scripts, images. Failure to do so creates a vector for man in the middle attack and also causes the browser to display a mixed SSL warning message.

Don't Allow HTTP Access to Secure Pages

- Never provide an authenticated page or a login page over HTTP. HTTPS should be used for the login landing page and all subsequent authenticated pages.
- The most secure approach is to display a warning when a user requests the HTTP page to instruct the user to bookmark or type the HTTPS page for future use. However, the more common approach is to just redirect from the HTTP request to the HTTPS equivalent page.

More info on SSL/TLS design can be found here

Implement STS

Where possible, we should utilize STS headers.

Content Security Policy (CSP)

Develop sites without inline JavaScript so adoption of CSP is easier

https://developer.mozilla.org/en/Introducing_Content_Security_Policy

Logging

See Security/Users_and_Logs

Admin Login Pages

The following are generally blockers for any website using an admin page:

1. Controls are in place to prevent brute force attacks
Options (any of these are fine):

- Admin page behind ssl vpn (most popular option)
- Account Lockout
- CAPTCHA's after 5 failed logins
- IP restrictions for access to the admin page

2. The login page and all admin pages are exclusively accessed over HTTPS. Any attempts to access a HTTP page redirect to HTTPS

3. The session id uses the SECURE flag

4. The session id uses the HTTPOnly flag

Configuring Wordpress Admin Pages Securely

Uploads

Attacks of Concern: Malformed user uploads containing JavaScript, HTML or other executable code, Arbitrary file overwrite

General Uploads

Upload Verification

- Use input validation to ensure the uploaded filename uses an expected extension type
- Ensure the uploaded file is not larger than a defined maximum file size

Upload Storage

- Use a new filename to store the file on the OS. Do not use any user controlled text for this filename or for the temporary filename.
- Store all user uploaded files on a separate domain (e.g. mozillafiles.net vs mozilla.org). Archives should be analyzed for malicious content (anti-malware, static analysis, etc)

Public Serving of Uploaded Content

- Ensure the image is served with the correct content-type (e.g. image/jpeg, application/x-xpinstall)

Beware of "special" files

- The upload feature should be using a whitelist approach to only allow specific file types and extensions. However, it is important to be aware of the following file types that, if allowed, could result in security vulnerabilities.
- "crossdomain.xml" allows cross-domain data loading in Flash, Java and Silverlight. If permitted on sites with authentication this can permit cross-domain data theft and CSRF attacks. Note this can get pretty complicated depending on the specific plugin version in question, so its best to just prohibit files named "crossdomain.xml" or "clientaccesspolicy.xml".
- ".htaccess" and ".htpasswd" provides server configuration options on a per-directory basis, and should not be permitted. See http://en.wikipedia.org/wiki/Htaccess

Image Upload

Upload Verification

- Use image rewriting libraries to verify the image is valid and to strip away extraneous content.
- Set the extension of the stored image to be a valid image extension based on the detected content type of the image from image processing (e.g. do not just trust the header from the upload).
- Ensure the detected content type of the image is within a list of defined image types (jpg, png, etc)

Archive Uploads

Upload Verification

- Ensure that the decompressed size of each file within the archive is not larger than a defined maximum size
- Ensure that an uploaded archive matches the type expected (e.g. zip, rar, gzip, etc)
- For structured uploads such as an add-on, ensure that the hierarchy within the archive contains the required files

Error Handling

Attacks of Concern: Sensitive Information Disclosure, System Information Disclosure, Aiding exploitation of other vulnerabilities

User Facing Error Messages

Error messages displayed to the user should not contain system, diagnostic or debug information.

Debug Mode

Debug mode is supported by many applications and frameworks and is acceptable for Mozilla applications. However, debug mode should only be enabled in stage.

Formatting Error Messages

Error messages are often logged to text files or files viewed within a web browser.

- text based log files: Ensure any newline characters (%0A%0C) are appropriately handled to prevent log forging
- web based log files: Ensure any logged html characters are appropriately encoded to prevent XSS when viewing logs

Recommended Error Handling Design

- Log necessary error data to a system log file
- Display a generic error message to the user
- If necessary provide an error code to the user which maps to the error data in the logfile. A user reporting an error can provide this code to help diagnose the issue

Further Reading

- OWASP Top 10
- OWASP Cheat Sheets
- OWASP Guide Project
- Php Sec Library
- Django Security
- Ruby Security

Contributors

Michael Coates - mcoates [at] mozilla.com
Chris Lyon - clyon [at] mozilla.com
Mark Goodwin - mgoodwin [at] mozilla.com

Last revision (mm/dd/yy): **02/21/2016**

https://www.owasp.org/index.php/AJAX_Security_Cheat_Sheet

AJAX Security Cheat Sheet

Introduction

This document will provide a starting point for AJAX security and will hopefully be updated and expanded reasonably often to provide more detailed information about specific frameworks and technologies.

Client Side (Javascript)

Use .innerText instead of .innerHtml

The use of .innerText will prevent most XSS problems as it will automatically encode the text.

Don't use eval

Eval is evil, never use it. Needing to use eval usually indicates a problem in your design.

Canonicalize data to consumer (read: encode before use)

When using data to build HTML, script, CSS, XML, JSON, etc. make sure you take into account how that data must be presented in a literal sense to keep it's logical meaning. Data should be properly encoded before used in this manner to prevent injection style issues, and to make sure the logical meaning is preserved.

Check out the OWASP Encoding Project.

Don't rely on client logic for security

Least ye have forgotten the user controls the client side logic. I can use a number of browser plugging to set breakpoints, skip code, change values, etc. Never rely on client logic.

Don't rely on client business logic

Just like the security one, make sure any interesting business rules/logic is duplicated on the server side less a user bypass needed logic and do something silly, or worse, costly.

Avoid writing serialization code

This is hard and even a small mistake can cause large security issues. There are already a lot of frameworks to provide this functionality. Take a look at the JSON page for links.

Avoid building XML or JSON dynamically

Just like building HTML or SQL you will cause XML injection bugs, so stay way from this or at least use an encoding library or safe JSON or XML library to make attributes and element data safe.

- XSS (Cross Site Scripting) Prevention
- SQL Injection Prevention

Never transmit secrets to the client

Anything the client knows the user will also know, so keep all that secret stuff on the server please.

Don't perform encryption in client side code

Use TSL/SSL and encrypt on the server!

Don't perform security impacting logic on client side

This is the overall one that gets me out of trouble in case I missed something :)

Server Side

Protect against JSON Hijacking for Older Browsers

Use CSRF Protection

- Cross-Site Request Forgery (CSRF) Prevention

Always return JSON with an Object on the outside

Always have the outside primitive be an object for JSON strings:

Exploitable:

```
[{"object": "inside an array"}]
```

Not exploitable:

```
{"object": "not inside an array"}
```

Also not exploitable:

```
{"result": [{"object": "inside an array"}]}
```

Avoid writing serialization code. Remember ref vs. value types!

Look for an existing library that has been reviewed.

Services can be called by users directly

Even though you only expect your AJAX client side code to call those services the users can too. Make sure you validate inputs and treat them like they are under user control (because they are!).

Avoid building XML or JSON by hand, use the framework

Use the framework and be safe, do it by hand and have security issues.

Use JSON And XML Schema for Webservices

You need to use a 3rd party library to validate web services.

Authors and Primary Editors

Til Mas
Michael Eddington