OWASP Top 10

Doc Writer

Version 1.0.0-SNAPSHOT, 2014-09-09

Table of Contents

Cross Site Scripting . . . 1
Reflected . . . 1
  Example . . . 1
  Summary . . . 2
Persistent (Stored) XSS . . . 2
  Summary . . . 3
DOM XSS . . . 3
  Example . . . 4
  Summary . . . 4
Broken Authentication and Session Management . . . 4
Insecure TLS Validation . . . 4
Unvalidated Redirects and Forwards . . . 4
Cross Site Request Forgery . . . 4
Sensitive Data Exposure . . . 4
Security Misconfiguration . . . 5
Insecure Direct Object References . . . 5
Injection . . . 5
Missing Function Level Access Control . . . 5
Source Code . . . 5
Attributes . . . 5
Includes . . . 6
build.gradle . . . 6

This is a user manual for an example project.

Cross Site Scripting

Reflected

- Unlike Persistent XSS, with Reflected Cross-Site Scripting (XSS) attacker-supplied script code is never stored within the application itself.
  - Instead the attacker crafts a malicious request to the application to illicit a single HTTP response by the application that contains the attackers's supplied script code.
  - Successful attacks require victim users to open a maliciously crafted link (which is very easy to do).
- To help defend against Reflected Cross Site Scripting (XSS) attacks it is important to ensure that user-supplied data is output encoded before being served by the application to other users.
  - In essence output encoding effectively works by escaping user-supplied data immediately before it is served to users of the application.
  - By correctly escaping the data, when it is served to the user for display in their browser, the browser does not interpret it as code and instead interprets it as data, thus ensuring it does not get executed.
  - For example the string: `<script>` is converted to: `&lt;script&gt;` when properly escaped and simply render as text in the user's browser window, rather than being interpreted as code.

Example

- In order to generate the search results web page, TradeSEARCH developers have used the JSP Standard Templating Library (JSTL) which provides standard actions and methods for formatting and rendering HTML pages.
  - When rendering a user's search result web page, the c:when conditional tag is called to check if any search results were returned by the server, which are then formatted and rendered by searchResults.jsp
  - However, should no matching results be found for a specified keyword, the c:otherwise conditional tag will render the HTML markup on line 9 and 10 that renders a No results found for: message followed by the user supplied search phrase.
  - To render the No results found for: error message followed by our search phrase string, `request.getParameter()` method is called to extract the search parameter from the URL which then gets directly rendered as a JSP expression.
  - e.g. For projects?search=sometext the JSP expression `request.getParameter("search")` will yield sometext which is finally rendered in the user's browser.
  - Unfortunately, JSP's expression language does not escape expression values, so if the search parameter contained HTML formatted data, Java's EL expression will simply render this string without escaping or encoding it first.

    - Finally when the No results found for: message is displayed, the search parameters HTML data string will also get rendered, thereby allowing code injection in the users browser context.
- The most effective method to protect against XSS attacks is by using JSTL's c:out tag or `fn:escapeXml()` EL function when displaying user-controlled input. These tags will automatically escape and encode HTML characters within the rendered HTML including < , > , " , ' and & thereby preventing injection of potentially malicious HTML code.

Summary

- In this module we learned how Reflected Cross Site Scripting (XSS) attacks work.
- We also learned that Reflected XSS and Non-Persistent XSS are two different terms for the same thing.
- Furthermore, we understood that to protect against XSS, Output Encoding needs to take place whenever user-supplied data is used by the application when building responses to other users.
- Finally, we learned that the `<c:out>` function escapes HTML characters to help prevent against XSS.

Persistent (Stored) XSS

- Persistent Cross-Site Scripting (XSS) is an application vulnerability whereby a malicious user tricks a web application into storing attacker-supplied script code which is the later served to unsuspecting user(s) of the application.
    - The attacker-supplied script code runs on the client-side system of other end user(s) of the application.
    - This type of vulnerability is widespread and affects web applications that utilize (unvalidated) user-supplied input to generate (unencoded) application output, that is served to users.
- To help defend against Stored Cross Site Scripting (XSS) attacks it is important to ensure that user-supplied data is output encoded before being served by the application to other users.
    - In essence output encoding effectively works by escaping user-supplied data immediately before it is served to users of the application.
    - By correctly escaping the data, when it is served to the user for display in their browser, the browser does not interpret it as code and instead interprets it as data, thus ensuring it does not get executed.
    - For example the string: `<script>` is converted to: `&lt;script&gt;` when properly escaped and is simply rendered as text in the user's browser window, rather than being interpreted as code.
- The most effective method to protect against XSS attacks is by using JSTL's c:out tag or `fn:escapeXml()` EL function when displaying user-controlled input. These tags will automatically escape and encode HTML characters within the rendered HTML including < , > , " , ' and & thereby preventing injection of potentially malicious HTML code.

Summary

- In this module we learned how Persistent Cross Site Scripting (XSS) attacks work.
- We also learned that Persistent XSS and Stored XSS are two different terms for the same thing.
- Furthermore, we understood that to protect against XSS, Output Encoding needs to take place whenever user-supplied data is used by the application when building responses to other users.
- Finally, we learned that the `<c:out>` function escapes HTML characters to help prevent against XSS.

DOM XSS

- Document Object Model (DOM) Based XSS is a type of XSS attack wherein the attacker's payload is executed as a result of modifying the DOM "environment" in the victim's browser used by the original client side script, so that the client side code runs in an "unexpected" manner.
- That is, the page itself (the HTTP response that is) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment.
- To defend against Cross Site Scripting attacks within the application user's Browser Document Object Model (DOM) environment a defense-in-depth approach is required, combining a number of security best practices.
    - Note You should recall that for Stored XSS and Reflected XSS injection takes place server side, rather than client browser side.
    - Whereas with DOM XSS, the attack is injected into the Browser DOM, this adds additional complexity and makes it very difficult to prevent and highly context specific, because an attacker can inject HTML, HTML Attributes, CSS as well as URLs.
    - As a general set of principles the application should first HTML encode and then Javascript encode any user supplied data that is returned to the client.
    - Due to the very large attack surface developers are strongly encouraged to review areas of code that are potentially susceptible to DOM XSS, including but not limited to:
        - `window.name`
        - `document.referrer`
        - `document.URL`
        - `document.documentURI`
        - `location`
        - `location.href`
        - `location.search`
        - `location.hash`
        - `eval`
        - `setTimeout`
        - `setInterval`
        - `document.write`

    - `document.writeIn`
    - `innerHTML`
    - `outerHTML`

Example

- In our modified code example, an additional check is introduced which performs input validation against the name string variable.
- To accomplish this, we make use of javascript's `match()` method to run a regular expression search on name variable, identifying non alphanumeric characters e.g. # , < , > , " , ' , & .
- Should any non alphanumeric characters be encountered, the if check will fail and invoke the "Security error" warning, thus preventing malicious javascript or HTML characters from being passed to the `document.write()` method.

Summary

- In this module we learned how Document Object Model (DOM) Cross Site Scripting (XSS) attacks work, or DOM XSS for short.
- We also learned that writing code more securely to protect against DOM XSS is much trickier than Persistent XSS and Stored XSS, due to the much larger attack surface with DOM XSS.
- We also learned as a general rule an application should first HTML encode, and then Javascript encode user-supplied (or otherwise untrusted data) to improve an application's security posture against DOM XSS.
- Finally, we learned that as developers there are a number of areas of code that are potentially susceptible to DOM XSS and extra consideration should be made for XSS DOM vulnerabilities in these areas (innerHTML , outerHTML , eval , document.write etc).

Broken Authentication and Session Management

Insecure TLS Validation

Unvalidated Redirects and Forwards

Cross Site Request Forgery

Sensitive Data Exposure

Security Misconfiguration

Insecure Direct Object References

Injection

Missing Function Level Access Control

Source Code

Java code from project

```
public boolean contains(String haystack, String needle) {
    return haystack.contains(needle);
}
```

Attributes

Built-in

asciidoctor-version

1.5.5

safe-mode-name

unsafe

docdir

/Users/mateusz/Scripts/asciidoc/owasp-asciidoc/src/docs/asciidoc

docfile

/Users/mateusz/Scripts/asciidoc/owasp-asciidoc/src/docs/asciidoc/OWASP-Java.adoc

imagesdir

./images

Custom

project-version

1.0.0-SNAPSHOT

sourcedir

/Users/mateusz/Scripts/asciidoc/owasp-asciidoc/src/main/java

endpoint-url

http://example.org

Includes

include::subdir/_b.adoc[]

content from *src/docs/asciidoc/subdir/_b.adoc*.

include::_c.adoc[]

content from *src/docs/asciidoc/subdir/c.adoc*.

Includes can be tricky!

build.gradle

```
buildscript {
    dependencies {
        classpath 'org.asciidoctor:asciidoctorj-pdf:1.5.0-alpha.15'
    }
}

plugins {
    id 'org.asciidoctor.convert' version '1.5.8.1'
}

apply plugin: 'java'

version = '1.0.0-SNAPSHOT'

asciidoctorj {
    version = '1.5.5'
}

asciidoctor {
    backends 'pdf'
    attributes \
        'build-gradle': file('build.gradle'),
        'sourcedir': project.sourceSets.main.java.srcDirs[0],
        'endpoint-url': 'http://example.org',
        'source-highlighter': 'coderay',
        'imagesdir': './images',
        'toc': 'left',
        'icons': 'font',
        'setanchors': '',
        'idprefix': '',
        'idseparator': '-',
        'docinfo1': ''
}
```