# CRV 2.0 Peer Review/Change Journal

Please show Contributor/Reviewer name and date time for each change

## Table of contents

1. Larry Conklin 3/1/16 12:35 PM

Remove hex 5 from each A1 thru A10 in the table of contents. It is enough to have a page saying section 5 with A1 to A10 following it in the guide.

Table of contents should be something like

Section 5 OWASP Top 10 Vulnerabilities

A1

A2

…

A10

## Methodology section

1. Larry Conklin 3/1/16 12:46 PM

Page 16 Table 9 should be table 10. Table 10 lists common free and commercial static analysis tools and features/languages support.

Make sure list of tables in table of contents is correct.

1. Missing figure. A threat tree as shown in figure 2(???) is useful to perform such threat analysis.
2. Missing table. To apply STRIDE to the flow diagram items the following table can be used(???).
3. Missing content. Detail examples of how to carry out threat modeling is given in appendix ???.
4. Missing appendix and make sure content is in appendix. Appendix ??? gives particle examples of how to carry code crawling in the following programming languages.

OWASP Top 10 Vulnerabilities section 5

1. Add the following content to end of content to A1.

# Client side JavaScript

JavaScript has several known security vulnerabilities, with HTML5 and JavaScript becoming more prevalent in web sites today and with more web sites moving to responsive web design with its dependence on JavaScript the code reviewer needs to understand what vulnerabilities to look for. JavaScript is fast becoming a significant point of entry of hackers to web application. For that reason we have included in the A1 Injection sub section.

The most significant vulnerabilities in JavaScript are cross-site scripting (XSS) and Document Object Model, DOM-based XSS.

Detection of DOM-based XSS can be challenging. This is cause by the following reasons.

* JavaScript is often obfuscated to protect intellectual property.
* JavaScript is often compressed out of concerned for bandwidth.

In both of these cases it is strongly recommended the code review be able to review the JavaScript before it has been obfuscated and or compressed. This is a huge point of contention with QA software professionals because you are reviewing code that is not in its production state.

Another aspect that makes code review of JavaScript challenging is its reliance of large frameworks such as Microsoft .Net and Java Server Faces and the use of JavaScript frameworks, such as JQuery, Knockout, Angular, Backbone. These frameworks aggravate the problem because the code can only be fully analyzed given the source code of the framework itself. These frameworks are usually several orders of magnitude larger then the code the code reviewer needs to review.

Because of time and money most companies simple accept that these frameworks are secure or the risks are low and acceptable to the organization.

Because of these challenges we recommend a hybrid analysis for JavaScript. Manual source to sink validation when necessary, static analysis with black-box testing and taint testing.

First use a static analysis. Code Reviewer and the organization needs to understand that because of event-driven behaviors, complex dependencies between HTML DOM and JavaScript code, and asynchronous communication with the server side static analysis will always fall short and may show both positive, false, false –positive, and positive-false findings.

Black-box traditional methods detection of reflected or stored XSS needs to be preformed. However this approach will not work for DOM-based XSS vulnerabilities.

Taint analysis needs to be incorporated into static analysis engine. Taint Analysis attempts to identify variables that have been 'tainted' with user controllable input and traces them to possible vulnerable functions also known as a 'sink'. If the tainted variable gets passed to a sink without first being sanitized it is flagged as vulnerability.

Second the code reviewer needs to be certain the code was tested with JavaScript was turned off to make sure all client sided data validation was also validated on the server side.

Code examples of JavaScript vulnerabilities.

<html>

<script type=”text/javascript”>

var pos=document.URL.indexOf(“name=”)+5;

document.write( document.URL.substring(pos,document.URL.length));

</script>

<html>

Explanation: An attacker can send a link such as “<http://hostname/welcome.html#name=><script>alert(1)</script> to the victim resulting in the victim’s browser executing the injected client-side code.

var url = document.location.url;

var loginIdx = url.indexOf(‘login’);

var loginSuffix = url.substring(loginIdx);

url = ‘[http://mySite/html/sso/’](http://mysite/html/sso/%E2%80%99) + loginSuffix;

document.location.url = url;

Line 5 may be a false-positive and prove to be safe code or it may be open to “Open redirect attack” with taint analysis the static analysis should be able to correctly identified if this vulnerability exists. If static analysis relies only on black-box component this code will have flagged as vulnerable requiring the code reviewer to do a complete source to sink review.

## Additional examples and potential security risks

Source: document.url

Sink: document.write()

Results: document.write(“<script>malicious code</script>

Cybercriminal may controlled the following DOM elements including…document.url,document.location,document.referrer,window.location

Source: document.location

Sink: windon.location.href

Results: windon.location.href = [http://www.BadGuysSite](http://www.badguyssite/); - Client code open redirect.

Source: document.url

Storage: windows.localstorage.name

Sink: elem.innerHTML

Results: elem.innerHTML = <value> =Stored DOM-based Cross-site Scripting

eval() is prone to security threats, and thus not recommended to be used.

Consider these points:

1. Code passed to the eval is executed with the privileges of the executer. So, if the code passed can be affected by some malicious intentions, it leads to running malicious code in a user's machine with your website's privileges.
2. A malicious code can understand the scope with which the code passed to the eval was called.
3. You also shouldn’t use eval() or new Function() to parse JSON data.

The above if used may raise security threats. JavaScript when used to dynamically evaluate code will create a potential security risk.

eval('alert("Query String ' + unescape(document.location.search) + '");');

eval(untrusted string); Can lead to code injection or client-side open redirect.

JavaScripts “new function” also may create a potential security risk.

## Three points of validity are required for JavaScript

1. Have all the logic server-side, JavaScript validation be turned off on the client
2. Check for all sorts of XSS DOM Attacks, never trust user data, know your source and sinks
3. Check for insecure JavaScript libraries and update them frequently.

## References:

http://docstore.mik.ua/orelly/web/jscript/ch20\_04.html

https://www.owasp.org/index.php/Static\_Code\_Analysis

http://www.cs.tau.ac.il/~omertrip/fse11/paper.pdf

<http://www.jshint.c>om

1. Add the following content to end of JavaScript in A1

# JSON (JavaScript Object Notation)

JSON (JavaScript Object Notation )is an open standard format that uses easy to read text to transmit data between a server and web applications. JSON data can be used by a large number of programming Languages and is becoming the de-facto standard in replacing XML.

JSON main security concern is JSON text dynamically embedded in JavaScript, because of this injection is a very real vulnerability. The vulnerability in the program that may inadvertently to run a malicious script or store the malicious script to a database. This is a very real possibility when dealing with data retrieved from the Internet.

The code reviewer needs to make sure the JSON is not used with Javascript eval. Make sure JSON.parse(…) is used.

Var parsed\_object = eval(“(“ + Jason\_text + “)”); // Red flag for the code reviewer.

JSON.parse(text[, reviver]); .. // Much better then using javascript eval function.

Code reviewer should check to make sure the developer is not attempting to reject known bad patterns in text/string data, Using regex or other devices is fraught with error and makes testing for correctness very hard. Allow only whitelisted alphanumeric keywords and carefully validated numbers.

Do not allow JSON data to construct dynamic HTML. Always us safe DOM features like innerText or CreateTextNode(…)

1. Add content below JSON

# Object/Relational Mapping (ORM).

Object/Relation Mapping (ORM) facilitates the storage and retrieval of domain objects via HQL (Hibernate Query Language) or .Net Entity framework.

It is a very common misconception that ORM solutions, like hibernate are SQL Injection proof. They are not. ORM’s allow the use of "native SQL". Thru proprietary query language, called HQL is prone to SQL Injection and the later is prone to HQL (or ORM) injection. Linq is not sql and because of that is not prone to sql injection. However using excutequery or excutecommand via linq causes the program not to use linq protection mechanism and is vulnerability to SQL injection.

## Bad Java Code Examples

List results = session.createQuery("from Items as item where item.id = " + currentItem.getId()).list();

NHibernate is the same as Hibernate except it is an ORM solution for Microsoft .Net platform. NHibernate is also vulnerable to SQL injection if used my dynamic queries.

## Bad .Net Code Example

string userName = ctx.GetAuthenticatedUserName();

String query = "SELECT \* FROM Items WHERE owner = '"

+ userName + "' AND itemname = '"

+ ItemName.Text + "'";

List items = sess.CreateSQLQuery(query).List()

## Code Reviewer Action

Code reviewer needs to make sure any data used in an HQL query uses HQL parameterized queries so that it would be used as data and not as code.