A1 Injection

This vulnerability covers SQL, LDAP, Xpath, OS commands, XML parsers.

Sql injection

Possible Impact:
1. Disclosure/leaking of sensitive information.
2. Data integrity issues. SQL injection may modify data, add new data, or delete data.
3. Elevation of privileges.
4. Gaining access to back-end network.

Using string concatenation to generate a SQL statement is very common in legacy applications where de- velopers were not considering security. The issue is this coding technique does not tell the parser which part of the statement is code and which part is data. In situations where user input is concatenated into the SQL statement, an attacker can modify the SQL statement by adding SQL code to the input data.

Untrusted input : (defense in depth)
1. HtmlEncode all user input.
2. Using static analysis tools.
3. Parameterize SQL queries (prepared statments):
Use SQL methods provided by the programming language or framework that parameterize the
statements, so that the SQL parser can distinguish between code and data.
4. Use Stored Procedures:
Stored procedures will generally help the SQL parser differentiate code and data. However Stored
Procedures can be used to build dynamic SQL statements allowing the code and data to become
blended together causing the it to become vulnerable to injection.

You should never use string concatenation in combination with the client input value

SQL Injection in PHP remains the number one attack vector, and also the number one reason for data compromises
Example :

```
<?php
$pass=$_GET[“pass”];
$con = mysql_connect(‘localhost’, ‘owasp’, ‘abc123’);
mysql_select_db(“owasp_php”, $con);
$sql=”SELECT card FROM users WHERE password = ‘”.$pass.”’”; 5
$result = mysql_query($sql);
?>
```

Addslashes :
You will avoid Sql injection using addslashes() only in the case when you wrap the query string with quotes.

mysql_real_escape_string():
mysql_real_escape_string() is a little bit more powerful than addslashes() as it calls MySQL’s library function mysql_real_escape_string, which prepends backslashes to the following characters: \x00, \n, \r, \, ‘, “ and \x1a.
As with addslashes(), mysql_real_escape_string() will only work if the query string is wrapped in quotes.

Parameter collections
Parameter collections such as SqlParameterCollection provide type checking and length validation. If you use a parameters collection, input is treated as a literal value, and SQL Server does not treat it as executable code, and therefore the payload can not be injected.

Using a parameters collection lets you enforce type and length checks. Values outside of the range trigger an exception. Make sure you handle the exception correctly. Example of the SqlParameterCollection:
Hibernate Query Language (HQL)
Hibernate facilitates the storage and retrieval of Java domain objects via Object/Relational Mapping (ORM).

```
1 using (SqlConnection conn = new
SqlConnection(connectionString)) {
2 DataSet dataObj = new DataSet();
3 SqlDataAdapter sqlAdapter = new SqlDataAdapter( “StoredProc”,
conn); sqlAdapter.SelectCommand. CommandType =
4 CommandType.StoredProcedure;
5 sqlAdapter.SelectCommand.Parameters.Add(“@usrId”,
SqlDbType.VarChar, 15);
6 sqlAdapter.SelectCommand.Parameters[“@usrId“].Value=UID.Text;
```

What to Review

- Always validate user input by testing type, length, format, and range.
- Test the size and data type of input and enforce appropriate limits.
- Test the content of string variables and accept only expected values. Reject entries that contain binary data, escape sequences, and comment characters.
- When you are working with XML documents, validate all data against its schema as it is entered.
- Never build SQL statements directly from user input.
- Use stored procedures to validate user input, when not using stored procedures use SQL API provided by platform. i.e. Parameterized Statements.
- Implement multiple layers of validation.
- Never concatenate user input that is not validated. String concatenation is the primary point of entry for script injection.
- You should review all code that calls EXECUTE, EXEC, any SQL calls that can call outside resources or command line.

JSON (JavaScript Object Notation)

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

```
Varparsed_object=eval(“(“+Jason_text +“)”); //
Redflagforthecodereviewer.
JSON.parse(text[, reviver]); .. // Much better then using javascript
eval function.
```

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 CreateText- Node(...)

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.

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 vulnera- bility to SQL injection.

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.

Content Security Policy (CSP)

Is a W3C specification offering the possibility to instruct the client browser from which location and/or which type of resources are allowed to be loaded. To define a loading behavior, the CSP specification use "directive" where a directive defines a loading behavior for a target resource type. CSP helps to detect and mitigate cer- tain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware

Directives can be specified using HTTP response header (a server may send more than one CSP HTTP header field with a given resource representation and a server may send different CSP header field values with dif- ferent representations of the same resource or with different resources) or HTML Meta tag, the HTTP headers below are defined by the specs:

• Content-Security-Policy : Defined by W3C Specs as standard header, used by Chrome version 25 and later, Firefox version 23 and later, Opera version 19 and later.
• X-Content-Security-Policy : Used by Firefox until version 23, and Internet Explorer version 10 (which partially implements Content Security Policy).
• X-WebKit-CSP : Used by Chrome until version 25

Risk

The risk with CSP can have 2 main sources:
• Policies misconfiguration,
• Too permissive policies.

Table 11: Security Related HTTP Headers

Header name	Description	Example
Strict-Transport-Security https://tools.ietf.org/html/rfc6797	HTTP Strict-Transport-Security (HSTS) enforces secure (HTTP over SSL/TLS) connections to the server. This reduces impact of bugs in web applications leaking session data through cookies and external links and defends against Man-in-the-middle attacks. HSTS also disables the ability for user's to ignore SSL negotiation warnings.	Strict-Transport-Security: max-age=16070400; includeSubDomains
X-Frame-Options https://tools.ietf.org/html/draft-ietf-websec-x-frame-options-01 **Frame-Options** https://tools.ietf.org/html/draft-ietf-websec-frame-options-00	Provides Click jacking protection. Values: deny - no rendering within a frame, sameorigin - no rendering if origin mismatch, allow-from: DOMAIN - allow rendering if framed by frame loaded from DOMAIN	X-Frame-Options: deny
X-XSS-Protection [http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx X-XSS-Protection]	This header enables the Cross-site scripting (XSS) filter built into most recent web browsers. It's usually enabled by default anyway, so the role of this header is to re-enable the filter for this particular website if it was disabled by the user. This header is supported in IE 8+, and in Chrome (not sure which versions). The anti-XSS filter was added in Chrome 4. Its unknown if that version honored this header.	X-XSS-Protection: 1; mode=block

X-Content-Type-Options https://blogs.msdn.microsoft.com/ie/2008/09/02/ie8-security-part-vi-beta-2-update/	The only defined value, "nosniff", prevents Internet Explorer and Google Chrome from MIME-sniffing a response away from the declared content-type. This also applies to Google Chrome, when downloading extensions. This reduces exposure to drive-by download attacks and sites serving user uploaded content that, by clever naming, could be treated by MSIE as executable or dynamic HTML files.	X-Content-Type-Options: nosniff
Content-Security-Policy, X-Content-Security-policy,X-WebKit-CSP https://www.w3.org/TR/CSP/	Content Security Policy requires careful tuning and precise definition of the policy. If enabled, CSP has significant impact on the way browser renders pages (e.g., inline JavaScript disabled by default and must be explicitly allowed in policy). CSP prevents a wide range of attacks, including Cross-site scripting and other cross-site injections.	Content-Security-Policy: default-src 'self'
Content-Security-Policy-Report_Only https://www.w3.org/TR/CSP/	Like Content-Security-Policy, but only reports. Useful during implementation, tuning and testing efforts.	Content-Security-Policy-Report-Only: default-src 'self'; report-uri http://loghost.example.com/reports.jsp

Input validation

Regular expressions can be used to validate user input, but the more complicated the regular express are the more chance it is not full proof and has errors for corner cases. Regular expressions are also very hard fro QA to test. Regular expressions may also make it hard for the code reviewer to do a good review of the regular expressions.

Input validation

Whitelist --> blacklist
business login

Canonicalization

Canonicalization is the process by which various equivalent forms of a name can be resolved to a single stan- dard name, or the "canonical" name.

Bad example

```
public static void main(String[] args) {
      File x = new File("/cmd/" + args[1]);
      String absPath = x.getAbsolutePath();
}
```

Good example

```
public static void main(String[] args) throws IOException {
      File x = new File("/cmd/" + args[1]);
      String canonicalPath = x.getCanonicalPath();
}
```

.NET Request Validation

One solution is to use .Net "Request Validation". Using request validation is a good start on validating user data and is useful. The downside is too generic and not specific enough to meet all of our requirements to provide full trust of user data.
You can never use request validation for securing your application against cross-site scripting attacks.

to determine whether the Uri provided by a user is valid.

```
var isValidUri = Uri.IsWellFormedUriString(passedUri,
UriKind.Absolute);

var uriToVerify = new Uri(passedUri);
var isValidUri = uriToVerify.IsWellFormedOriginalString();
var isValidScheme = uriToVerify.Scheme == "http" ||
uriToVerify.Scheme == "https";
```

Before rendering user input as HTML or including user input in a SQL query, encode the values to ensure malicious code is not included.

You can HTML encode the value in markup with the <%: %> syntax, as shown below.

```
<span><%: userInput %></span>
```

Or, in Razor syntax, you can HTML encode with

```
<span>@userInput</span>

var encodedInput = Server.HtmlEncode(userInput);
```

Managed Code and Non-Managed Code

Both Java and .Net have the concept of managed and non-managed code. To offer some of these protections during the invocation of native code, do not declare a native method public. Instead, declare it

private and expose the functionality through a public wrapper method. A wrapper can safely perform any necessary input validation prior to the invocation of the native method:
Java Sample code to call a Native Method with Data Validation in place

```
public final class NativeMethodWrapper {
	private native void nativeOperation(byte[] data, int offset, int len);

	public void doOperation(byte[] data, int offset, int len) {
		// copy mutable input
		data = data.clone();
		// validate input
		if(offset<0||len<0||offset>data.length-len){
			throw new IllegalArgumentException();
		}
		nativeOperation(data, offset, len); }
	}
}
```

Data validations checklist for the Code Reviewer.

• Ensure that a Data Validation mechanism is present.
• Ensure all input that can (and will) be modified by a malicious user such as HTTP headers, input fields, hidden fields, drop down lists, and other web components are properly validated.
• Ensure that the proper length checks on all input exist.
• Ensure that all fields, cookies, http headers/bodies, and form fields are validated.
• Ensure that the data is well formed and contains only known good chars if possible.
• Ensure that the data validation occurs on the server side.
• Examine where data validation occurs and if a centralized model or decentralized model is used.
• Ensure there are no backdoors in the data validation model.
• “Golden Rule: All external input, no matter what it is, will be examined and validated.”