A Gap Analysis of Application Security in Struts2/WebWork

Arshan Dabirsiaghi
Project Lead, OWASP Intrinsic Security Working Group
Director of Research, Aspect Security

Special thanks to the following contributors:

Nick Coblentz
Andrew van der Stock

May 4, 2009

Abstract

The purpose of this paper is to discover what features and capabilities, if any, the Struts2/WebWork (hereafter referred to simply as Struts2) development team could add to increase the security of applications built with Struts2. The version analyzed was version 2.1.6, which was the latest version available when the project was started. The purpose of this research is not to discover security flaws within Struts2, but rather to discover how the Struts2 framework allows developers to build security into applications, and how that process can be improved.

The only non-commercial application security library that covers all necessary security is an expert tool called Enterprise Security API (ESAPI), which is maintained by OWASP (http://owasp.org/). Throughout the paper there will be many comparisons to the ESAPI project since it serves as a de-facto model of security best practices. Acegi Security, recently renamed to Spring Security, is a framework that allows for excellent enterprise authentication and authorization configuration, but lacks coverage of other areas covered by ESAPI.

Table of Contents

Introduction

The research is broken down into several logical categories in order to manage the scope and easily deliver partial and consumable results:

- Authentication
- Access Control (aka Authorization)
- Input Validation
- Cross-Site Scripting (output encoding)
- Cross-Site Request Forgery
- Error Handling

Rather than give a large number of gap data points and forcing a high level summary at the end of the document, which will inevitably end up being not appropriately descriptive, each section will have its own conclusions on coverage and a general set of recommendations.

It is not the purpose of the paper to denigrate the quality of the Struts2 project, whose many qualities have played a large part in its popularity and longevity in a field where frameworks trend into and out of popularity very quickly. Rather, the purpose of the paper is to discover ways to improve the security of applications developed with Struts2 through framework enhancements. Also, it is recognized by the author that the framework must accommodate developers who want to build applications that have low-value assets and should therefore not be forced to invest the necessary time to implement higher assurance security. Or, more plainly, if an application is not that important, Struts2 should not force developers to implement an inappropriate amount of security.

We also recognize that Struts2 does not attempt to cover all the security areas mentioned in this paper. The Struts2 development team most likely acknowledges that you use log4j to fulfill your logging needs and they don't attempt to step outside of their domain of expertise by reinventing already available well-built mechanisms. However, exactly this type of "one stop shopping" is necessary for development teams in order to avoid the complexity of managing an application's security as decentralization of the security grows.

Another core principal in the analysis is the ease of use of a security mechanism. One of the tenets of security API is that it must be easy to understand and use. If it is not, it will be called incorrectly, haphazardly, or not at all (ASPECT1).

The attacks, vulnerabilities and countermeasures discussed in this paper can be found in various OWASP sources, including the OWASP Guide (OWASP1) to Building Secure Web Applications, the OWASP Testing Guide (OWASP7), and the OWASP Code Review Guide (OWASP8).

Centralized Security Mechanisms

The architecturally centralized components for performing security checks within Struts2 must be invoked by extended base Action classes or Interceptor classes which must be explicitly added to referenced Interceptor chains.

After some thinking on this strategy, the question that remains is this: what value does the Struts2 Interceptor add as opposed to the servlet filter? In simple terms, they are both custom Java that get executed before a subset of requests. Therefore, at a high level it is easy to say that they can both do the same thing. However, is the deployment strategy and development of one easier or safer to accomplish security goals with?

Real world J2EE applications often include a number of architecturally "one-off" servlets or endpoints. These endpoints accomplish cross-application requirements like performing redirects, logging users off of SSO systems, serving images, etc. Therefore, any mechanism that only protects Struts2-based resources that are likely to cause vulnerabilities in applications. There is more assurance (catch all resource requests instead of just Struts2) and equivalent deployment work (insert an entry into an XML file) in creating a J2EE filter instead of a Struts2 interceptor or extended base Action.

The following centralized security mechanisms are accomplishable in Struts2, but are similarly easier and more safely implemented in one or multiple J2EE filters. For more information on implementing these features in Struts2 for comparison purposes, consult the appendix:

- Authentication (see Appendix A)
- URL access control (also called "page-level" access control (OWASP3)) (see Appendix B)
- Anti-caching controls (see Appendix C)
- CSRF controls (see Appendix D)
- Uncaught exception handling (see Appendix E)
- Enforcing SSL requirements (see Appendix F)
- Regenerating session IDs (see Appendix G)

The rest of the security areas discussed in the paper are those areas that are required to be decentralized, or those areas that the Struts2 framework attempts to accomplish and therefore require discussion. There are also some security areas that were not covered in the paper. The reasons they were not included include one or many of the following:

- The security area is not thought to be solvable by an external library
- The security area is not close enough to the goal of Struts2 to warrant inclusion

- The problem area doesn't usually include vulnerabilities that rank above a "low" in common threat assessment standards

The security areas that met one or more of those criteria are as follows:

- Error handling patterns (avoiding fail-open, exception swallowing, information leakage, resource exhaustion)
- Logging (how to log, when to log, what to log)
- Cryptography (having a simple-to-use API, encouraging strong algorithms, protecting the right data)
- Backend communication (connecting with least privilege, channel protection, resource exhaustion)

Authentication

The Struts2 framework doesn't contain provide any authentication mechanisms directly to implement. It does provide "interceptors" which are much like traditional J2EE servlet filters, but are closer architecturally to Struts2 as opposed to the container. These appear to be the most architecturally reliable places to put authentication protections (enforcing authentication *after* login). So, that leaves the developer who has authentication requirements to do one of the following:

1. Rely on container-based authentication
2. Use custom interceptors in common stacks in order to control access to Actions that require authentication
3. Use traditional J2EE servlet filters in order to control access to URLs that require authentication

It is very uncommon to find any medium to large enterprise applications that rely on container-based authentication because of the lack of flexibility this option offers. For small, internal applications it is considered sufficient if security requirements and principals are not expected to change much over the lifetime of the application.

A typical enterprise security architecture calls for applications to be protected by authentication. Most of the time, this is within a certain URL structure, like "/app". There will also be a certain number of unprotected pages, like the login form, that need to be listed as exceptions. Given the discussion in the previous section, it is considered more secure to use a J2EE servlet in order to implement authentication.

User Management in Struts2

There is no idea of "user management" in the Struts2 framework. There is no base "User" class or any kind of static utilities in order to facilitate easy user administration. Because of the lack of inborn user management capabilities and complementary API, developers must compensate with custom code or 3rd party libraries against the following attacks that are strictly related to authentication:

- Username harvesting
- Brute forcing
- Massive account lockouts

Improvements in Authentication

There are a number of improvements that could be made to the Struts2 framework in managing the safe authentication and maintenance of users.

Provide an Authentication Architecture

Although JAAS is available in any J2EE application, it is widely unused. It's not fine-grained enough for some organizations, it's too fine-grained for others, and it's not easy to understand. Because of this, JAAS will only fit at some organizations. Therefore, it is recommended that Struts2 implement its own authentication mechanism.

In order to attain the coverage that's needed in enterprise applications, the authentication mechanism should preempt any other servlet filter in order to allow developers to protect Struts2 and non-Struts2 resources.

Provide Static Utilities for Authentication

There are a number of static utilities concerning authentication that are often written by developers who are not security experts, and are therefore not usually written correctly. A set of static utilities that should be made available to developers in a framework include the following:

- A function that can determine password strength
- A secure password generator
- A simple password hashing function

The following table shows what ESAPI and Struts2 offer in the area of authentication:

Function	ESAPI	Struts2
Has Authentication Mechanism Built-In	**Yes**	**X**
Authentication Covers All Application Resources	**Yes**	**X**
Secure User Interface Class	**Yes**	**X**
Secure User Management Functions	**Yes**	**X**
API for Password Strength Measurement	**Yes**	**X**
API for Secure Password Generation	**Yes**	**X**
API for Simple Password Hashing	**Yes**	**X**

Session Management

By default, J2EE application servers have a very simple session management lifecycle: if the application server receives a request without a valid session ID, the application server will issue them a new one. Also, if the difference in elapsed time between any two requests is greater than some inactivity limit, then the session is terminated.

However, there are other security concerns around how sessions are implemented that are not addressed by J2EE, and thus must be addressed at a framework or application level. Struts2 does not influence how the J2EE application servers create, manage, or destroy sessions, so the following attacks must be compensated for with custom application code:

- Session fixation
- URL-rewriting
- Cookie disclosure through XSS
- Cookie exposure through sniffing
- Permanent session hijacking

Improvements to Session Management

There are a number of improvements that could be made to the Struts2 framework, both in managing the lifecycle of sessions and in protecting the session cookies themselves from disclosure.

Disabling URL Rewriting or Rotate Session IDs at Application Boundaries

Application servers use URL-rewriting in order to allow cookie-less session management. In practice, this means application servers usually rewrite URLs on a new session's first response to contain the session identifier. This way, if the user's next request comes in without a session cookie, the application server will still be able to track their state by parsing the token out of the URL.

This feature can be used by attackers to seed a victim's session ID by sending the victim a link with a session ID in the URL. After the victim logs into the application under the auspices of the attacker's session ID, the attacker can make requests to the application and be recognized as the victim.

This attack can be mitigated in one of two ways. First, the application framework can prevent URL-rewriting from being used. This is not ideal since some users will not be able to use the application and business may be lost. The more acceptable solution is to force a rotation of session ID values when crossing application boundaries.

For instance, when a user "logs in", the application should rotate the session ID so that an attacker that knew or set the user's session ID won't know the new session ID, and therefore won't have any means of presenting themselves to the application server as the victim. The implementation of the "rotation" should involve invalidating the old session and creating a new session so that no victim-specific data is in

any session except the new, safe session that will be delivered on the response to the request which contained valid credentials.

This capability would require some way for users to mark certain Actions as security boundaries. A very natural place to do that would be in the configuration file where Actions are defined. Such a marking could allow applications to perform other security actions, like requiring re-authentication.

Add Absolute Timeout Capability

There is often a business requirement that applications not permit sessions to exist, despite activity level, for more than some amount of time. This is different from an idle timeout, which application servers all support and are used to kill sessions that have been left idle for too long. This "absolute timeout" serves many purposes, including limiting the scope of damage of a stolen session and making sure users who have been removed from an enterprise application can't sustain their access by never allowing the idle session timeout from occurring with traffic automation.

It would be beneficial to the security of enterprise applications built with Struts2 to offer a framework-level absolute timeout mechanism that is configurable by the developer.

Add Enhanced Cookie Protection Options

There are two security-relevant cookie flags that are honored by the major browsers. The *secure* flag tells the browser not to send the cookie over an unencrypted channel. Sites that use a mix of SSL and non-SSL pages typically don't enable this flag because non-SSL requests will appear to be unauthenticated users since they won't be accompanied by the authenticated session cookie. The *HttpOnly* flag tells the browser that client-side JavaScript should not have access to the cookie on which the flag was set. A cookie value with both of these flags set would look something like the following:

```
Set-Cookie: JSESSIONID=abc123; domain=foo.com; HttpOnly=; secure=;
```

There is no way in the J2EE specification or in the various implementations to programmatically set the *secure* or *HttpOnly* flags on the application server cookie. It is possible in some implementations to configure the server to set these two flags. However, in most implementations this is not possible.

It should be noted that it is possible to set the *secure* flag, and in a few cases the *HttpOnly* flag, on custom cookies created by the application.

Because of the lack of adoption, consistency and clarity surrounding the *secure* and *HttpOnly* flags, many applications can't or don't take advantage of the critical protections these flags offer.

Therefore, it would be beneficial to Struts2 developers if there was an easy configuration option to allow developers to enable "enforce SSL" and "protect session cookie from XSS" features. The implementation details may require submitting a session ID as a form parameter (MANICODE1) or a 3-step HTTP handshake (OWASP9) between the client and server. However, the implementation details of these features are irrelevant as long as the goal of these protections is accomplished.

The following table shows what ESAPI and Struts2 offer in the area of session management:

Function	ESAPI	Struts2
API for Rotating Session ID	Yes	X
Automatically Rotate Session ID on Login	X	X
Framework Disables URL Rewriting	Yes	X
Allows Programmatically Setting HttpOnly on Custom Cookies	Yes	X
Allows Adding HttpOnly/Secure Flags to Session Cookies	Yes	X
Automatic Absolute Timeout Handling	X	X

Programmatic Access Control

For most enterprise applications, there is a need for creating fine grained access control systems that execute after URL-based access control occurs. There are many URLs in applications that are accessible by multiple roles; therefore further distinction regarding allowed access must be made before business case execution.

Also, there is usually a need for managing access to certain rows or records. This is called “data layer” access control, because similarly privileged users’ data must be segmented. For instance, user A should not be able to see the paycheck of user B, even though they are both in the ‘employee’ role. A universal model for controlling access to object instances is considered difficult to design because each application’s requirements for modeling data are different. The ESAPI project has a general-purpose set of interfaces to make the implementation of data-layer access easier with an existing framework.

There are no functional or data-layer access control methods or architecture delivered as part of Struts2. Current standard practices in the J2EE development space include utilizing Acegi, JAAS, ESAPI, creating a custom architecture, or not doing anything at all if security is not a concern. Because no access control mechanisms are provided, developers must write custom code or utilize 3rd party libraries to compensate for the following attacks:

- Direct object reference attacks (OWASP2)
- Privilege escalation

Improvement in Programmatic Access Control

A model for all the necessary dimensions of a properly granular access control mechanism can be found in the ESAPI project (OWASP4). A wholesale import of such functionality is in order if access control is a goal of the Struts2 development team. The following table shows what functions ESAPI and Struts2 offer in the area of access control:

Function	ESAPI	Struts2
API for Access Control Check on URL	**Yes**	**X**
API for Access Control Check on Functions	**Yes**	**X**
API for Access Control Check on Data	**Yes**	**X**
API for Access Control Check on File	**Yes**	**X**

API for Access Control Check on Service	**Yes**	**X**

Input Validation

There are many different places to perform input validation. How effective they are depends on how much context the validation mechanism has about the use of the parameter. Therefore, the closer the application's validation is to the actual use of the parameter, the better the application can decide if the value is appropriate.

This means that a web application firewall or J2EE filter can't do proper input validation unless it is supplied validation rules for each URL and parameter. This type of system, as was found in OWASP Stinger and in many custom implementations, is generally thought not to be ideal since it requires constant maintenance of a data file that describes all the validation. Over time, managing this data file becomes too cumbersome a process and eventually the data file becomes ignored or out of date. This is the type of input validation strategy in use in Struts2.

In order to validate a request for an Action, the developer must create a **<ActionClassName>-validation.xml** file that contains the rules for that specific Action. A sample validation XML file may look like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
<validators>
    <field name="name">
        <field-validator type="requiredstring">
            <message>You must enter a name</message>
        </field-validator>
    </field>
    <field name="age">
        <field-validator type="int">
            <param name="min">13</param>
            <param name="max">19</param>
            <message>Only people ages 13 to 19 may take this quiz</message>
        </field-validator>
    </field>
</validators>
```

This validation file validates two things: that a parameter called "name" is present and that a parameter called "age" is an integer between 13 and 19. It is a tedious process to add or change validation rules for a parameter because of the XML bloat.

Improvements in Input Validation

There are a few improvements that could be done to increase the usability, security and efficiency of input validation in the Struts2 framework.

Bring Back the validate() Method

As far as general improvements are considered, an alternative to the data-driven validation is to enforce a validation pattern in the code, as was possible in Struts1. It is still possible however that an abstract validation method required to be created by the architecture can simply be stubbed out to return success and perform no validation, effectively foregoing any security. However, the developer must undertake a conscious effort to ignore security in bypassing this kind of architecture.

Provide a More Secure and Usable API for Getting Parameters

Accompanying the enforced-validate() approach should be a system whereby developers can't access a raw parameter value without somehow specifying validation. For example, the only way to access a parameter is through an API like the following:

```
someHttpRequestWrapper.getParameter(String parameterName, IValidationRule rule)
someHttpRequestWrapper.getParameter(String parameterName, List<IValidationRule> rules)
someHttpRequestWrapper.getParameter(String parameterName, String regexp)
```

These methods could be configured to throw an exception or return null if validation rules fail. Similar assert() type APIs could be provided if access to the parameter wasn't necessary where validation was taking place.

The ESAPI project has put a substantial amount of research into usable and reliable patterns in input validation (OWASP5). It is our suggestion that the Struts2 developers consider the input validation patterns in ESAPI and possibly integrate the features optionally before eventually phasing out older techniques.

Add Validation Annotations to Input POJOs

As far as improving the existing infrastructure, it would be easy and powerful if a user could put validation annotations inside the POJOs used as the input object model. Since these annotations are in the Action class, they require more work to tie to individual parameters.

Add Canonicalization API

Another major piece of input validation that is missing is canonicalization. ESAPI also has taken great strides in demonstrating to the J2EE community how to properly break down input into its simplest form in order to avoid encoding and fragmentation attacks in downstream consumers like Web Services, browsers, databases and more. Proper canonicalization is necessary for input validation.

Add More Default Validators

Finally, it would be good if Struts2 bundled more validators with the project. There is already a good set of validators now, including an email and URL validator. However, many real world applications run into other common validation problems, such as how to validate a relative URL or filename safely.

Many applications would also benefit from an input validation API that automatically checked to see if input received from the user was possible given the data delivered to them. For example, if a hidden field delivered to the user was modified, this should cause an error in the framework since it is possible to detect this problem automatically, assuming there's no JavaScript code that drastically alters input once received or creates new input forms that are not expected. This type of parameter-tampering prevention has been implemented and can be studied in the Secure Parameter Filter project (GOTHAM1).

Add a Workflow Scope

Another approach that can help application developers avoid hidden fields is the idea of a framework-level "Workflow" scope, which is used by the Groovy framework. For wizard or multi-stage form functionality, a developer can put information into a workflow-scoped object on the server side in order to remove the need for putting that data into hidden fields. Any step in a workflow will from that point forward have access to all of the data put into the workflow scope. The workflow scope is actually stored within the session, but the framework should automatically manage the data. The framework relays the information from the session when queried and it is cleared from the session when the workflow is terminated. The following table shows what ESAPI and Struts2 offer in the area of input validation:

The following table shows what ESAPI and Struts2 offer in the area of input validation:

Function	ESAPI	Struts2
Has Architectural Input Validation Option	X	Yes
API for Input Validation	Yes	Yes
Validate Method Enforce as Part of Action Framework	X	X
Input Validation Annotations on Action	X	Yes
Input Validation Annotations on Input Object Model	X	X
API for Canonicalization	Yes	X

Has Default Validators	**Yes**	**Yes**
Extended Set of Validators (HTML, file paths, etc.)	**Yes**	**X**
Has Workflow Scope	**X**	**X**

Cross-Site Scripting

There are generally two ways to defend against cross-site scripting: input validation and output encoding. Input validation is not considered safe by itself because some injection contexts require very few “special” characters in order to be exploitable. Also, an architecture that enforces strong input validation on every field is difficult to achieve both technically and as part of a software development process. Because of these circumstances, an emphasis on output encoding mechanisms available in a platform is due. There are a number of criteria on which we can decide the effectiveness of an encoding API:

- Does it support encoding based on HTML context (such as inside an attribute or JavaScript block)?
- Does the encoding algorithm properly encode all characters needed to prevent an “up” or “down” injection in the target context?
- Is it obvious when to use a particular mechanism?
- Can the mechanism produce output based on an arbitrary character set?

Output Mechanisms and Their Encoding Properties

Struts 2 JSP tags like <s:textfield/>, <s:textarea/>, and other UI tags help prevent basic XSS by encoding 3 very important special characters: ‘<’, ‘>’ and ‘”’.

The UI components, even with this limited encoding, are safe without a separate encoding vulnerability (e.g., when the application uses US-ASCII or UTF-7 or doesn’t specify one, or if the output encoding is controlled by the user). Therefore, a developer in Struts2 can use the default mechanisms available for the safely building web forms with the following objects:

- Text fields
- Text areas
- Hidden fields
- File fields
- Checkboxes

It is not very difficult to build an application using only these UI controls. Situations where inputs are used but <form> tags are not (like Ajax) are slightly awkward but still possible by using an empty Struts2 form tag need forms make things slightly awkward because an empty form is needed. However, the tools are in Struts2 to build secure UI components.

However, XSS can occur when user input appears anywhere on a page, and not necessarily in a UI component. According to the OWASP Prevention Cheatsheet, there are 5 injection contexts that need unique levels of protection.

The first place and one of the most common places where XSS occurs is in the text body between a start and end tag, like the following:

 [user input]

The expected tag to use in this situation would be <s:property/>, which is a generic output mechanism that encodes the same 3 characters as the UI tags: “<”, “>”, and “”’ (double quote). This is safe assuming that application controls the page encoding and doesn’t choose an unsafe one.

The second context to look at is also a common place for XSS to occur: inside HTML attributes. We have to breakdown the attributes into categories when talking about encoding – JavaScript event handlers and everything else. For “everything else”, a safe encoding mechanism needs to encode many things. The following is a quote from the Cheatsheet that talks about why so many characters need encoding for HTML attributes:

> ***Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the &#xHH; format (or a named entity if available) to prevent switching out of the attribute. The reason this rule is so broad is that developers frequently leave attributes unquoted. Properly quoted attributes can only be escaped with the corresponding quote. Unquoted attributes can be broken out of with many characters, including [space] % * + , - / ; < = > ^ and |.***

The <s:property/> tag only encodes the 3 character’s we’ve mentioned before, and obviously does not meet this criteria so it is not safe *in general* for encoding data that goes into non-event handler attributes.

The second set of attributes is JavaScript event handlers. Many times security folks will suggest that “HTML-encoding” output prevents XSS. While true in some contexts, HTML-encoding in JavaScript sections doesn’t mean much. The rules for encoding there are vastly different from that of a normal HTML context. Encoding data that goes into JavaScript (whether inside event handlers or script tags) is different:

> ***Except for alphanumeric characters, escape all characters less than 256 with the \xHH format to prevent switching out of the data value into the script context or into another attribute. Do not use any escaping shortcuts like \" because the quote character may be matched by the HTML attribute parser which runs first. If an event handler is quoted, breaking out requires the corresponding quote. The reason this rule is so broad is that developers frequently leave event handler attributes unquoted. Properly quoted attributes can only be escaped with the corresponding quote. Unquoted attributes can be broken out of with many characters including [space] % * + , - / ; < = > ^ and |. Also, a </script> closing tag will close a script block even though it is inside a quoted string because the HTML parser runs before the JavaScript parser.***

There is no Struts2 tag (or utility method, or anything) that performs JavaScript encoding. Again, the <s:property/> tag only encodes the 3 basic characters, so that offers very little protection against user

input inside of JavaScript contexts. The same can be said of any data that ends up inside stylesheet sections.

Because of the lack of diverse or comprehensive encoding methods available in the framework, developers must compensate with custom code or libraries against the following XSS attack vectors:

- JavaScript injection
- Unquoted or single-quoted attribute-based injection
- Stylesheet injection

Improvement in XSS Defense Mechanisms

We suggest that the Struts2 team implement tags that match ESAPI's contextual output tags. In order to provide the ability to output raw (and dangerous, if contents are influenced by the user) data so that developers can send un-encoded data to the browser, there can be an "encode=false" option.

We suggest that the new tags be introduced as soon as possible so that the process of deprecation and eventual removal can begin.

The following table shows what ESAPI and Struts2 offer in the area of XSS protection:

Function	ESAPI	Struts2
Safe UI Controls	**X**	**Yes**
API + View Tag for HTML Element Context Encoding	**Yes**	**Yes**
API + View Tag for HTML Attribute Context Encoding	**Yes**	**X**
API + View Tag for JavaScript Encoding	**Yes**	**X**
API + View Tag for CSS Property Encoding	**Yes**	**X**
API + View Tag for HTML URI Encoding	**Yes**	**X**

Cross-Site Request Forgery

Ignoring downstream solutions from the browser, an application developer's highest assurance solution for CSRF is secure form tokens, which are also called nonces. The qualities of a good token in a CSRF defense mechanism include the following:

- The token value must be universally unique to user
- The token value must be computationally difficult to guess
- The token's legitimacy must have a comparable lifetime to the user's session

The following code snippet from org.apache.struts2.util.TokenHelper.java shows how Struts2 double-submit tokens are built:

```
private static final Random RANDOM = new Random();
...
public static String setToken(String tokenName) {
    Map session = ActionContext.getContext().getSession();
    String token = generateGUID();
    try {
        session.put(tokenName, token);
    }
...
public static String generateGUID() {
    return new BigInteger(165, RANDOM).toString(36).toUpperCase();
}
```

This is bad CSRF token for several reasons:

- It doesn't use a cryptographic-strength pseudo-random number generated (like java.security.SecureRandom)
- The token value is not specific to the user
- Although the token is approximately 20 bytes (which is technically long enough), the token value is forced to be 36-bytes large in representation. Thus, the token's randomness is disproportionate to its length, possibly giving a false sense of security.

So, Struts2 double-submit tokens are not a good defense against CSRF unless the TokenHelper utility is sub-classed, made better, and hacked back into the bundled token interceptor framework.

Because of the lack of strong, token-based protection available in the framework, developers must compensate with custom code or libraries against the following attacks:

- CSRF
- JavaScript Hijacking (FORTIFY1)

Improvements in CSRF Defense

Provide an additional “secure” or “csrf_strength” flag to the token interceptor framework that allows developers to optionally absorb the performance penalty of generating cryptographically unique and personalized random numbers for double submit token values. This is the cheapest and easiest place to put a CSRF protection in the application ecosystem and it is backwards-compatible.

The following table shows what ESAPI and Struts2 offer in the area of CSRF protection:

Function	ESAPI	Struts2
API for Generating CSRF Token	**Yes**	**X**
API for Validating CSRF Token	**Yes**	**X**
Framework Anti-CSRF Solution	**X**	**X**

Appendix A: Authentication

The following Interceptor, **AuthenticationInterceptor.java**, shows how to create a centralized authentication interceptor.

```
package com.nickcoblentz.struts2.interceptors;

import java.util.Collections;
import java.util.Map;
import java.util.Set;

import org.apache.struts2.StrutsStatics;

import com.nickcoblentz.rbaexample.security.AuthenticationAware;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
import com.opensymphony.xwork2.util.TextParseUtil;

public class AuthenticationInterceptor extends AbstractInterceptor {

   private String authenticationSessionField = new String("authenticated");
   private static final String authenticationRequiredResult =
"authentication_required";
   private Set excludeActions = Collections.EMPTY_SET;

   @Override
   public String intercept(ActionInvocation invocation) throws Exception {
      Map session = invocation.getInvocationContext().getSession();
      String actionName = invocation.getProxy().getActionName();

      if(invocation.getAction() instanceof AuthenticationAware) {
        action = ((AuthenticationAware) invocation.getAction());
        action.setActionsWithoutAuthentication(excludeActions);
      }

      Object authenticationObject = session.get(authenticationSessionField);

      if(excludeActions.contains(actionName) ||
        (authenticationObject!=null && authenticationObject instanceof Boolean &&
         authenticationObject.equals(Boolean.TRUE))) {
           return invocation.invoke();
      }
```

```
       return authenticationRequiredResult;
   }

   public void setAuthenticationSessionField(String authenticationSessionField) {
      this.authenticationSessionField = authenticationSessionField;
   }

   public void setExcludeActions(String values) {
      if (values != null) {
        this.excludeActions = TextParseUtil.commaDelimitedStringToSet(values);
      }
   }
}
```

The application would then have to reference the class as part of an interceptor chain as shown in the following **struts-security.xml**:

```
<?xml version="1.0" encoding="UTF-8" ?>

<struts>
      <constant name="struts.custom.i18n.resources" value="global-messages login-
messages" />

    <package name="loginPackage" namespace="/" extends="struts-security">
    ...
    <!--  The authentication interceptor ensures users have authenticated
          before allowing access to Actions unless the action in listed
          in the "excludeActions"  -->


    ...
            <interceptors>
                  <interceptor name="authenticationInterceptor"
class="com.nickcoblentz.struts2.interceptors.AuthenticationInterceptor" />

                  <interceptor-stack name="defaultSecurityStackWithAuthentication">
                        <interceptor-ref name="defaultSecurityStack" />
                        <interceptor-ref name="authenticationInterceptor">
                              <param
name="excludeActions">TokenError,SSLError,AuthenticationError,AuthorizationError,Custo
mError,Login,ProcessSimpleLogin</param>
                        </interceptor-ref>

                  </interceptor-stack>
```

```
            </interceptors>

            <default-interceptor-ref name="defaultSecurityStackWithAuthentication" />

      <global-results>
                  <result type="chain"
name="authentication_required">AuthenticationError</result>
            </global-results>

            ...

            <!-- Error Handling Actions.
             They should be excluded from authentication and authorization -->

      <action name="AuthenticationError"
class="com.nickcoblentz.rbaexample.actions.CustomError">
            <param name="errorMessage">error.authentication</param>
            <result>/WEB-INF/pages/security/CustomError.jsp</result>
       </action>
            <!-- Authentication Actions
                    They should be excluded from authentication and authorization -->
      <action name="Login">
            <result type="tiles" name="input">Login.simpleLogin</result>
            <result type="tiles">Login.simpleLogin</result>
      </action>

            <action name="ProcessSimpleLogin"
class="com.nickcoblentz.rbaexample.actions.ProcessSimpleLogin">
            <result name="input" type="chain">Login</result>
            <result type="redirect-action" name="success">Internal</result>
      </action>


       <!-- Actions Requiring authentication -->

            <action name="Internal">
               <result type="tiles">Authenticated.internal</result>
            </action>
   </package>

</struts>
```

Appendix B: URL-based Access Control

There is no built-in role-based access control mechanism protecting Actions in Struts2. Struts1 applications had this capability by assigning "roles" to actions defined in **struts-config.xml**. The standard method for accomplishing this capability is by creating a customer Interceptor. An example of this technique is displayed in the following **RolesInterceptor.java** class:

```
package com.nickcoblentz.struts2.interceptors;

import java.util.Collections;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.StringTokenizer;

import com.nickcoblentz.rbaexample.security.RolesAware;

import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
import com.opensymphony.xwork2.util.TextParseUtil;

public class RolesInterceptor extends AbstractInterceptor {

	private String roleSessionField = "role";
	private Map<String, Set> roleActions = Collections.EMPTY_MAP;
	private static final String AuthorizationRequiredResult =
"authorization_required";

	@Override
	public String intercept(ActionInvocation invocation) throws Exception {
		final String actionName = invocation.getProxy().getActionName();
		Map session = invocation.getInvocationContext().getSession();

		if(invocation.getAction() instanceof RolesAware) {
			RolesAware action = (RolesAware) invocation.getAction();
			action.setRoleActions(roleActions);
		}

		Object userRole = session.get(this.roleSessionField);
```

```
            if(hasSufficientRole(userRole, actionName))
                  return invocation.invoke();
            return AuthorizationRequiredResult;
      }

      public void setRoleActions(String roleActionsParam) {
            StringTokenizer roleActionsParamTokenizer = new
StringTokenizer(roleActionsParam,";");
            this.roleActions=new HashMap<String,
Set>(roleActionsParamTokenizer.countTokens());
            while(roleActionsParamTokenizer.hasMoreTokens()) {
                  String roleActionArray[] =
roleActionsParamTokenizer.nextToken().trim().split(":");
                  if(roleActionArray.length == 2) {
                        String role = roleActionArray[0].toLowerCase();
                        Set actions =
TextParseUtil.commaDelimitedStringToSet(roleActionArray[1]);
                        this.roleActions.put(role,actions);
                  }
            }
      }

      public boolean hasSufficientRole(Object userRole, String actionName) {
            if(roleActions.containsKey("*") &&
roleActions.get("*").contains(actionName))
                  return true;
            if(userRole !=null && userRole instanceof String) {
                  String userRoleString = ((String)userRole).toLowerCase();

                  if(roleActions.containsKey(userRoleString) &&
                              roleActions.get(userRoleString).contains(actionName))
                        return true;
            }
            return false;
      }
      public String getRoleSessionField() {
            return roleSessionField;
      }
      public void setRoleSessionField(String roleSessionField) {
            this.roleSessionField = roleSessionField;
      }
}
```

This technique, as was highlighted in the “Centralized Security Mechanisms” section, fails to protect any resource that isn’t a Struts2 Action.

Appendix C: HTTP Caching Headers

One way web applications can reduce the likelihood of browsers disclosing sensitive data through caching is to include HTTP headers within the server's response that tell the browser not to cache the received data.

```
Cache-control: no-cache, no-store
Pragma: no-cache
Expires: -1.
```

Depending on the browser, version and other circumstances, the “must-revalidate” flag and other such values must also be present. These headers can be included within a Struts2 application by creating a custom Interceptor. An example Interceptor class, **CachingHeadersInterceptor.java**, has been provided below.

```
package com.nickcoblentz.struts2.interceptors;

import javax.servlet.http.HttpServletResponse;
import org.apache.struts2.StrutsStatics;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;

public class CachingHeadersInterceptor extends AbstractInterceptor {

  public String intercept(ActionInvocation invocation) throws Exception {

    final ActionContext context = invocation.getInvocationContext();

    final HttpServletResponse response = (HttpServletResponse)
context.get(StrutsStatics.HTTP_RESPONSE);
    if(response!=null) {
      response.setHeader("Cache-control","no-cache, no-store");
      response.setHeader("Pragma","no-cache");
      response.setHeader("Expires","-1");
    }

    return invocation.invoke();
  }
}
```

Once a custom interceptor has been created, it must be included within the Struts2 Interceptor stack.

The example **struts-security.**xml below defines an abstract Struts package; this package must be extended before it can be used in an application.

```
<?xml version="1.0" encoding="UTF-8" ?>

<struts>
  <package name="struts-security" abstract="true" extends="struts-default">

    <interceptors>
      <interceptor name="cachingHeadersInterceptor"
class="com.nickcoblentz.struts2.interceptors.CachingHeadersInterceptor" />

      <interceptor-stack name="defaultSecurityStack">
        <interceptor-ref name="defaultStack" />
        <interceptor-ref name="cachingHeadersInterceptor" />
      </interceptor-stack>
    </interceptors>

    <default-interceptor-ref name="defaultSecurityStack" />
  </package>
</struts>
```

Appendix D: Cross-Site Request Forgery

The following **struts-security.xml** file demonstrates creating an Interceptor stack that includes a token check.

```
<?xml version="1.0" encoding="UTF-8" ?>

<struts>
    <package name="struts-security" abstract="true" extends="struts-default">
    <interceptors>
      <interceptor-stack name="defaultSecurityStack">
        <interceptor-ref name="defaultStack" />
        <interceptor-ref name="tokenSession">
          <param name="excludeMethods">*</param>
        </interceptor-ref>
      </interceptor-stack>
    </interceptors>
    <default-interceptor-ref name="defaultSecurityStack" />
    </package>
</struts>
```

The Interceptor stack must be referenced from your application's Struts2 configuration:

```
<action name="ProcessSimpleLogin"
class="com.nickcoblentz.example.actions.ProcessSomeAction">
  <result name="input" type="chain">SomeFailedAction</result>
  <result type="redirect-action" name="success">SomeSuccessfulAction</result>
  <interceptor-ref name="defaultSecurityStackWithAuthentication">
    <param name="tokenSession.includeMethods">*</param>
  </interceptor-ref>
</action>
```

The only thing left is to use the <s:token /> tag in a form that requires protection:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>

<s:actionmessage label="Messages"/>
<s:actionerror label="Errors"/>
<s:fielderror label="Field Errors" />
```

```
<s:form action="ProcessSimpleLogin" validate="true">
       <s:textfield name="username" label="Username" />
       <s:password name="password" label="Password" />
       <s:token />
       <s:submit value="Login" />
</s:form>
```

Appendix E: Handling Exceptions

Struts2 includes an Exception Interceptor in its default stack. Developers can utilize this interceptor to catch errors and redirect users to a page containing a generic error message. One example is shown below. The example **struts-security.xml** below defines an abstract Struts2 package; this package must be created before it can be used in an application.

```
<?xml version="1.0" encoding="UTF-8" ?>

<struts>
  <constant name="struts.custom.i18n.resources" value="global-messages" />

    <package name="struts-security" abstract="true" extends="struts-default">

    <global-results>
      <result type="chain" name="custom_error">CustomError</result>
    </global-results>

    <global-exception-mappings>
      <exception-mapping exception="java.lang.Exception" result="custom_error" />
    </global-exception-mappings>

        <action name="CustomError"
class="com.nickcoblentz.rbaexample.actions.CustomError">
          <result>/WEB-INF/pages/security/CustomError.jsp</result>
        </action>

    </package>
</struts>
```

Appendix F: Requiring SSL

One way to redirect non-SSL requests within Struts2 is to create an Interceptor to verify this connection. An example Interceptor, **RequireSSLInterceptor.java**, has been provided below.

```
package com.nickcoblentz.struts2.interceptors;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts2.StrutsStatics;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;

public class RequireSSLInterceptor extends AbstractInterceptor {

  public String intercept(ActionInvocation invocation) throws Exception {

    final ActionContext context = invocation.getInvocationContext();
    final HttpServletRequest request = (HttpServletRequest)
context.get(StrutsStatics.HTTP_REQUEST);

    if(request.isSecure()) {
      return invocation.invoke();
    }
    else {
      // invalidate session
      // redirect to a different page
    }
  }
}
```

It must be added to a referenced Interceptor stack as shown in the previous examples. The example **struts-security.xml** below defines an abstract Struts2 package that includes the Interceptor in the **defaultSecurityStack**; this package must be extended before it can be used in an application.

```
<?xml version="1.0" encoding="UTF-8" ?>

<struts>
  <package name="struts-security" abstract="true" extends="struts-default">

    <interceptors>
      <interceptor name="requireSSLInterceptor"
class="com.nickcoblentz.struts2.interceptors.RequireSSLInterceptor" />
```

```
      <interceptor-stack name="defaultSecurityStack">
      <interceptor-ref name="defaultStack" />
      <interceptor-ref name="requireSSLInterceptor" />
      </interceptor-stack>
    </interceptors>

    <default-interceptor-ref name="defaultSecurityStack" />

  </package>
</struts>
```

Appendix G: Session ID Regeneration

When users cross an authentication boundary, their session ID should be regenerated. This means that a user in an unauthenticated context should have a completely different JSESSIONID once they log in. This concept can also be applied to events such as an elevation in privileges, addition or removal of a role, a logout action, or a rotation between SSL and non-SSL requests.

In Struts2, a user's JSESSIONID can be regenerated using the following code:

```
package nickcoblentzblog.actions.sessions;
import java.util.Map;
import org.apache.struts2.interceptor.SessionAware;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.dispatcher.SessionMap;

public class Login extends ActionSupport implements SessionAware  {
  private String userid;
  private String password;
  private Map session;

  public String execute() {
    if(//check for successful authentication here)  {
      ((SessionMap)this.session).invalidate();
      this.session = ActionContext.getContext().getSession();

      session.put(//Mark the user's session "Authenticated" here);

      return SUCCESS;
    }
    else
      return ERROR;
  }

  public void setSession(Map session) {
    this.session = session;
  }
}
```

Bibliography

OWASP1. (n.d.). *Guide to Building Secure Web Applications*. Retrieved from OWASP.org: http://www.owasp.org/index.php/Category:OWASP_Guide_Project

OWASP2. (n.d.). *Top 10 2007-Insecure Direct Object Reference*. Retrieved from OWASP.org: http://www.owasp.org/index.php/Top_10_2007-A4

OWASP3. (n.d.). *Top 10 2007-Failure to Restrict URL Access*. Retrieved from OWASP.org: http://www.owasp.org/index.php/Top_10_2007-A10

OWASP4. (n.d.). *AccessController.java*. Retrieved from code.google.com: http://code.google.com/p/owasp-esapi-java/source/browse/trunk/src/org/owasp/esapi/AccessController.java?r=303

OWASP5. (n.d.). *Validator.java*. Retrieved from code.google.com: http://code.google.com/p/owasp-esapi-java/source/browse/trunk/src/org/owasp/esapi/Validator.java

OWASP6. (n.d.). ESAPI Canonicalization. Retrieved from jazoon.com: http://jazoon.com/download/presentations/5320.pdf

GOTHAM1. (n.d.). *Secure Parameter Filter (SPF)*. Retrieved from owasp.org: http://www.owasp.org/images/8/8f/SPF.pdf

OWASP7. (n.d.). *Testing Guide.* Retrieved from owasp.org: http://www.owasp.org/index.php/Category:OWASP_Testing_Project

OWASP8. (n.d.). *Code Review Guide*. Retrieved from owasp.org: http://www.owasp.org/index.php/Code_Review_Guide_Frontispiece

OWASP9. (n.d.). *ESAPIWebApplicationFirewallFilter.java*. Retrieved from owasp.org: http://code.google.com/p/owasp-esapi-java/source/browse/branches/1.5.wafexperiment/src/main/java/org/owasp/esapi/filters/waf/ESAPIWebApplicationFirewallFilter.java?r=437

FORTIFY1. (n.d.) *JavaScript Hijacking*. Retrieved from fortify.com: http://www.fortify.com/landing/downloadLanding.jsp?path=%2Fpublic%2FJavaScript_Hijacking.pdf

MANICODE1. (n.d.). *double-submit cookie CSRF defense and HTTPOnly*. Retrieved from blogspot.com: http://manicode.blogspot.com/2009/02/double-submit-cookie-csrf-defense-and.html

ASPECT1. (n.d.). *Breaking the Waterfall Mindset of the Security Industry*. Retrieved from owasp.org: http://www.google.com/url?sa=t&source=web&ct=res&cd=1&url=http%3A%2F%2Fwww.owasp.org%2F

images%2Fb%2Fb8%2FAppSecEU08-Agile_and_Secure.ppt&ei=YPDsSfDeLpn4Mcz5nNkF&usg=AFQjCNEG5Chb9JV38bGBjCsE_ZldX0Xevw