Security features of ZK Framework

This document provides a brief overview of security concerns related to JavaScript powered enterprise web application in general and how ZK built-in features secures ZK powered enterprise web applications and alleviates these security concerns.

Overview

Web application security is an important topic especially in the context of enterprise web applications. Due to increasing number of new web technologies with their new weaknesses being introduced and progress made by attackers, threat landscape for web applications security constantly changes. However there are some common threats that can be avoided easily. Some of them can be protected against by server configuration and some at application level. Further to that ZK framework has some built-in security features to protect against some common threats as described below

Few words on Javascript and Security

In today's web applications landscape Javascript is ubiquitous and so are security concerns related to it. Javascript, by nature, is exposed on the client side and interpreted by the browser. In ZK's context, although ZK UI component widgets are rendered and controlled by Javascript on client-side, *the business logic of the application is never exposed on the client-side*. This makes it impossible for attacker to reverse engineer ZK web application implementation and exploit any application features.

Now let's look at ZK Security features and how they counter security threats

Block requests for inaccessible widgets

With the advent of browser developer/debugging tools it is possible to inspect HTML rendered on the client-side and live edit them in real time. For example a hostile user can make a invisible button visible and trigger unexpected actions. Ideally it is recommended to not create such a widget in the first place. In other words it is recommended not to render the widget on client side instead of making it visible

```
<button unless="${accessible}"/>
<button visible="${accessible}"/>
```

First statement is safer than the second one as in first case the button html element will not be rendered on the client side in the first place.

However if not rendering widgets is not an option from application requirement point of view then you can enable **InaccessibleWidgetBlockService** block. **InaccessibleWidgetBlockService** block will filter out unwanted requests from inaccessible widgets whose client-side state might have been modified using debugging/developer tools.

You can enable **InaccessibleWidgetBlockService** block by specifying a listener in WEB-INF/zk.xml as shown below

```
<listener>
    <listener-class>org.zkoss.zkmax.au.InaccessibleWidget
BlockService$DesktopInit</listener-class>
</listener>
```

In many cases, you may only want to block particular events, not all events. For example, you may want to receive onOpen when a menupopup is going to show up. Then, you can specify a library property called IWBS.events to control the behavior of InaccessibleWidgetBlockService. For example,

```
<library-property>
    <name>org.zkoss.zkmax.au.IWBS.events</name>
    <value>onClick,onChange,onSelect</value>
</library-property>
```

Note : **InaccessibleWidgetBlockService** is availalble in ZK EE however it is very simple to implement your own block by implementing AuService interface. For example below implementation blocks only button component's onClick event

```
public class MyBlockService implements
org.zkoss.zk.au.AuService {
   public boolean service(AuRequest request, boolean
everError) {
       final Component comp = request.getComponent();
       return (comp instanceof Button) &&
"onClick".equals(request.getCommand());
          //true means block
   }
```

Cross-site scripting (XSS) Protection

Cross-site scripting (XSS) is a type of computer security vulnerability typically found in web applications that enables malicious attackers to inject client-side script into web pages viewed by other users. Because HTML documents have a flat, serial structure that mixes control statements, formatting, and the actual content, any non-validated user-supplied data included in the resulting page without proper HTML encoding may lead to markup injection.

To prevent from XSS attack, ZK component encodes any value that might be input by an user, such as the value of label and textbox, by escaping & and other unsafe characters. For example, the following statement is totally safe no matter what the value of any_value might be:

```
<textbox value="${any_value}"/>
```

However, there are still some notes worth to pay attention to.

The content Property of html and comboitem

The content property of the html and combitem components (Html.setContent(String) and Comboitem.setContent(String)) are designed to allow applications to generate HTML content directly. In other words, it is not encoded. Thus, it is better not to carry the value input by an user, unless it is encoded property. For example, if the value of `any_content` is, as in the following example, generated directly and vulnerable to XSS attack if it is the value provided by an user and without proper encoding.

```
<html>${any_content}</html>
```

Java API: `Html.setContent(String)` and `Comboitem.setContent(String)`

Denial-of-Sevice (DoS) Protection

From OWASP

> *The Denial of Service (DoS) attack is focused on making unavailable a resource (site, application, server) for the purpose it was designed. There are many ways to make a service unavailable for legitimate users by manipulating*

network packets, programming, logical, or resources handling vulnerabilities, among others. If a service receives a very large number of requests, it may stop providing service to legitimate users.
Denial-of-service attacks significantly degrade service quality experienced by legitimate users. It introduces large response delays, excessive losses, and service interruptions, resulting in direct impact on availability.

OWSAP also recommends a few techniques developers can employ against DoS at application level. You can read them here in more detail.

ZK has two features to prevent against DoS attack.

1. You can configure **max-desktops-per-session** and **max-requests-per-session** to avoid DoS attack by storing too much data in session or flooding application server with too many requests.

```
<session-config>
<max-desktops-per-session>a_number</max-desktops-per-sess
ion>
</session-config>
```

It indicates maximum number of desktops per session that are allowed. A desktop represents a HTML page for a browser. In other words, this number controls the number of concurrent browser windows allowed per session.

Note : A negative number means no limitation at all.

```
<session-config>
      <max-requests-per-session>a_number</max-requests
-per-session>
</session-config>
```

It indicates the maximum number of concurrent requests per session that are allowed. Each time a user types an URL at the browser, it creates a request and the request ends after the response is sent to the browser. In other words, this number controls how many concurrent requests the same user can send.

Note : A negative number means no limitation at all, but it is not recommended due to the possibility of the denial-of-service (DoS) attacks.

2. If your application has a button starting a long running operation you could use button.setAutodisable() in Java or autodisable="self" in ZUL from preventing DoS by repeatedly causing the button onClick which might hold up precious server resources. Please refer here for more details on autodisable for Button component

Additionally, you can enable autodisable for all button by metioning following in a custom language addon

```
<language-addon>
    <component>
        <component-name>button</component-name>
        <extends>button</extends>
        <property>
            <property-name>autodisable</property-name>
            <property-value>self</property-value>
        </property>
    </component>
</language-addon>
```

In addition to this, each and every ZK ajax request carries a additional http header called ZK-SID. The purpose of this ZK-SID header is to differentiate between multiple Ajax requests. If a same ajax request is resent with the same ZK-SID then it is ignored. This ZK-SID header helps to reduce server load and hence DoS attack by sending too many similar/repeated requests.

Cross-site Request Forgery (CSRF) Protection

From OWASP CSRF page

> *Cross-Site Request Forgery (CSRF) is an attack that tricks the victim into loading a page that contains a malicious request. It is malicious in the sense that it inherits the identity and privileges of the victim to perform an undesired function on the victim's behalf, like change the victim's e-mail address, home address, or password, or purchase something. CSRF attacks generally target functions that cause a state change on the server but can also be used to access sensitive data.*
>
> *For most sites, browsers will automatically include with such requests any credentials associated with the site, such as the user's session cookie, basic auth credentials, IP address, Windows domain credentials, etc. Therefore, if*

the user is currently authenticated to the site, the site will have no way to distinguish this from a legitimate user request.

In short, a successful CSRF attack uses a valid http request, but often with malicious data to cause unwanted and unintended results, which assumes a valid identity of end user by using above mentioned browser's way of handling of user related information.

ZK and CSRF attack limitations

According OWASP mentioned CSRF Limitations, in order to mount a successful CSRF attack several things have to happen

1. The attacker must target either a site that doesn't check the referrer header (which is common) or a victim with a browser or plug-in that allows referer spoofing (which is rare)

This can be avoided by adding a servlet filter that checks if all request referrer and origin headers contain the appropriate values

2. The attacker must find a form submission at the target site, or a URL that has side effects, that does something (e.g., transfers money, or changes the victim's e-mail address or password).

By design ZK is an Ajax solution. Because of this design generally no form submit nor specific URL request can cause side effects

3. The attacker must determine the right values for all the form's or URL's inputs; if any of them are required to be secret authentication values or IDs that the attacker can't guess, the attack will fail.

ZK generates *unique* ids for html elements that represent ZK components on client side and these unique ids are checked on server side when data containing them is passed via ZK's Ajax mechanism. For successful CSRF attack, the attacker will have to guess all unique ids for those html elements while submitting the malicious request. If the html element ids are not the same as they were when page rendered then the data is considered invalid by ZK and request is rejected at server side automatically.

Also note that ZK will regenerate these ids if the components are re-rendered via page refresh or component are re-created again.

4. The attacker must lure the victim to a Web page with malicious code while the victim is logged into the target site.

This is more of a humane issue and depends on the end user. Application developers should raise the awareness about CSRF by documenting this in their application documentation which end users can refer to.

ZK Desktop ID as CSRF token

General recommendation to prevent CSRF is to use ***Synchronizer Token Pattern***. Generally this is done by inserting a *unique* token usually referred as csrf token in the generated html and check it on server side on form submission.

ZK employs similar technique in the form of desktop id. Each url in ZK web application when requested gets associated a Desktop instance on server side. Please refer to Desktop and Pages for more details on the concept of Desktop in ZK.

When a page is rendered on client side a unique desktop id will also be rendered to client as shown in below image listing 1.

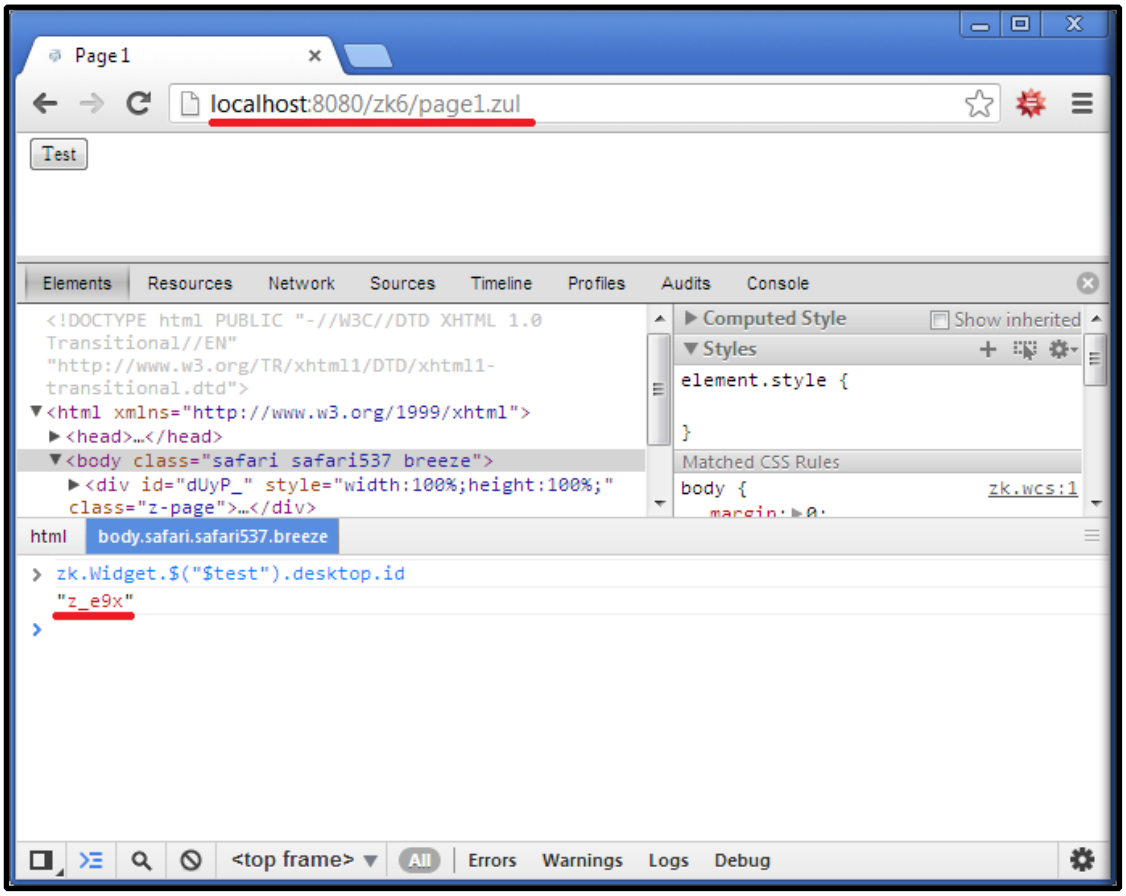

Image Listing 1: Image showing desktop id for page1.zul

ZK desktop is discarded and re-created each time a new page is loaded in browser or even the current page is refreshed. On each re-rendering of page a new automatically generated unique id is assigned to desktop as illustrated in below image listing 2 & 3

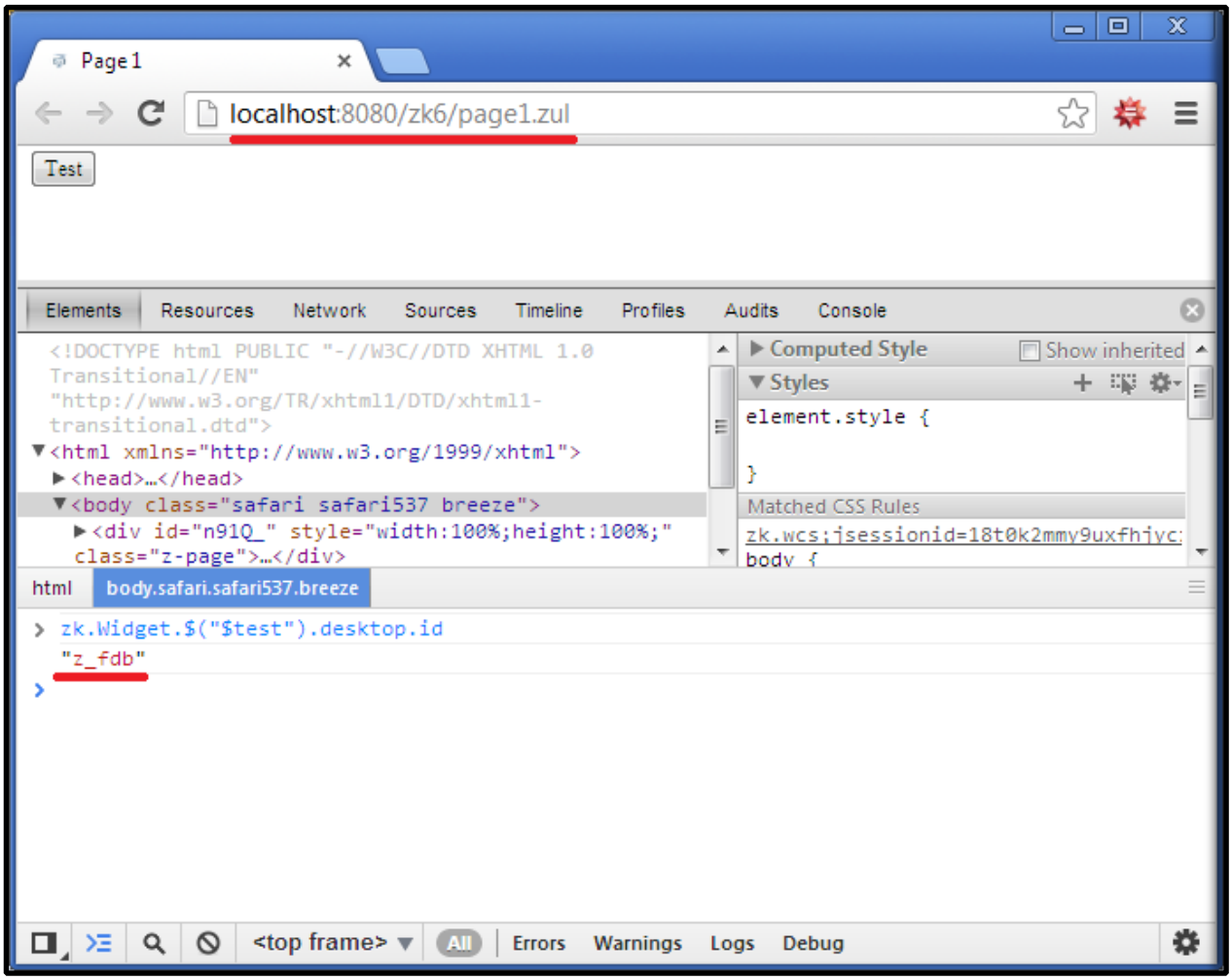

Image Listing 2: Image showing new desktop id for page1.zul on refresh/reload

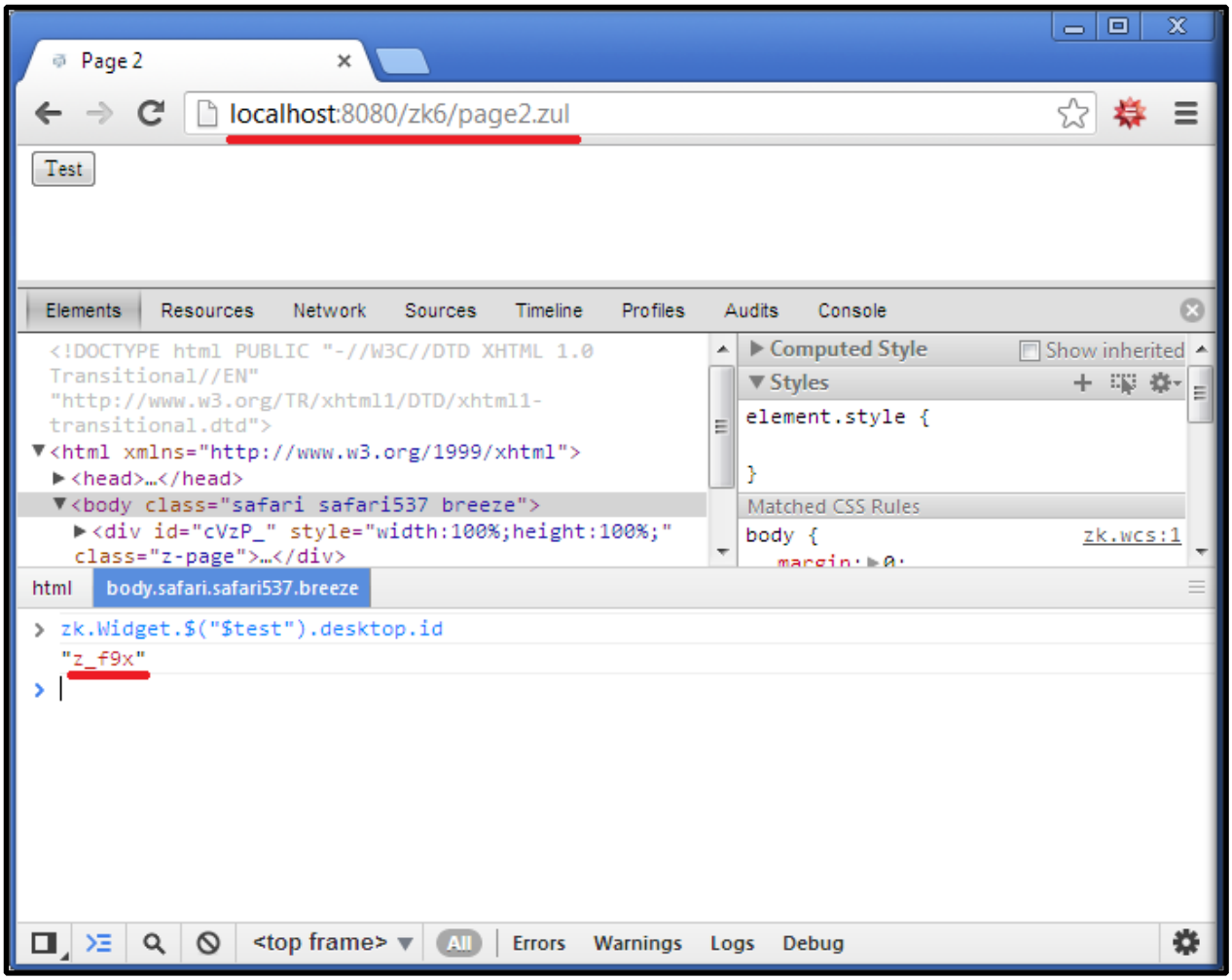

Image Listing 3: Image showing new desktop id for page2.zul

Once the page is loaded this desktop id is carried via ZK Ajax mechanism and on each interaction this unique desktop id will be passed as Ajax request POST data. Image listing 4 shows when a button is clicked on page ZK will send corresponding onClick event back to server as an Ajax request and this Ajax request carries the unique desktop id as POST data as shown in image listing 4

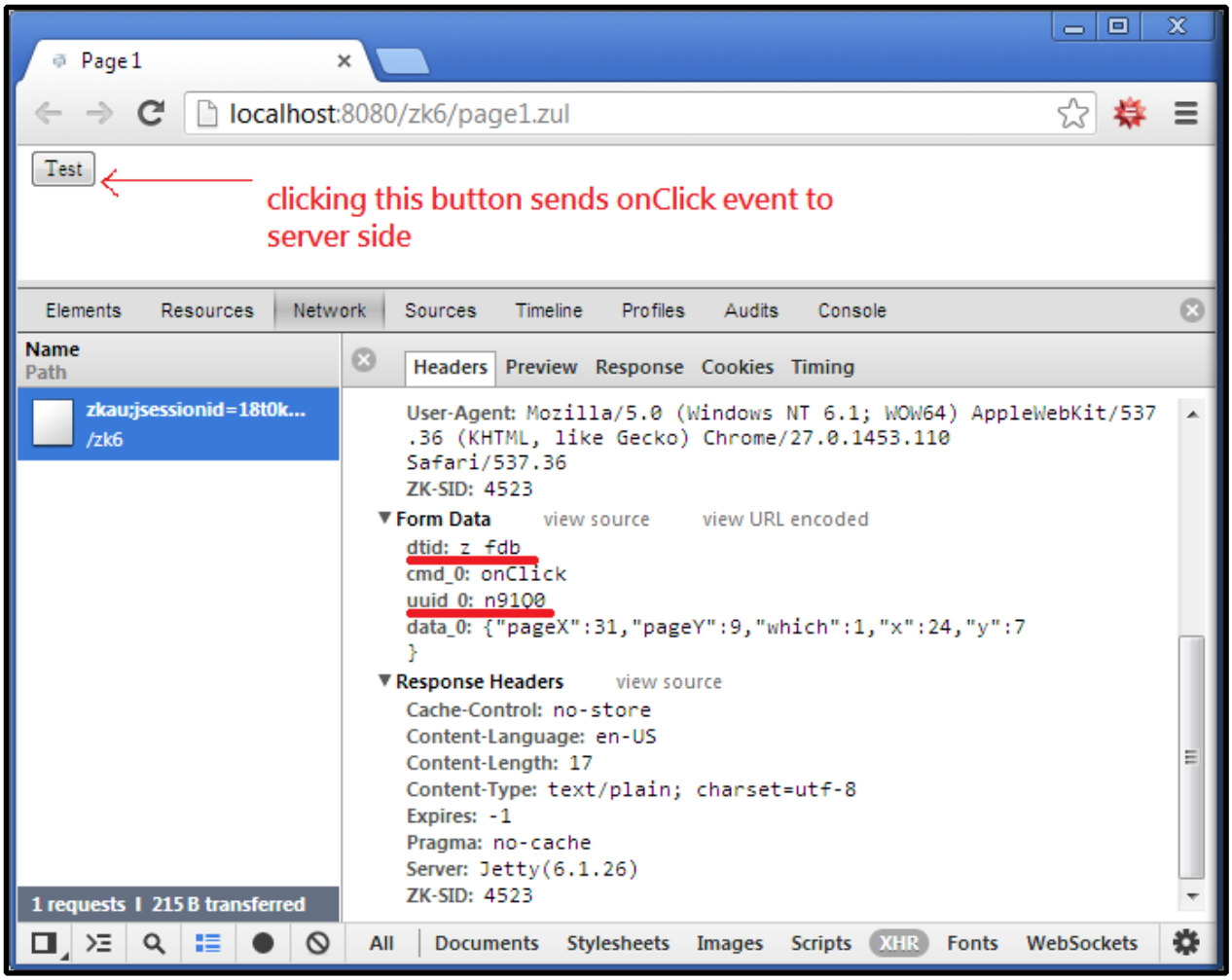

Image Listing 4: Image showing onClick event Ajax request containing unique desktop id as form data

ZK CSRF Protection Notes:

1. There is no One-to-One relation between Desktop id and http Session id
2. Desktop ID is unique per page per url. Even same url across different browser tabs in same browser instance will be assigned unique desktop ID.
3. For successful CSRF attack against, the attacker not only has to guess unique desktop ID but also each and every unique ids assigned to the html element for corresponding ZK widgets on client side. If even one is not correct then the entire request is rejected at server side without executing any application level code containing business logic.

About ZK

ZK is the leading enterprise Java Web framework with more than 1,500,000 downloads. ZK empowers a wide variety of companies and institutions, ranging from small to large in multiple industries. ZK's customers and users include Fortune Global 500 companies (such as Barclays, Sun Microsystems, Swiss Re, Oracle, Société Générale, Alcatel-Lucent, State Grid, China Southern Power Grid), governments and tens of thousands of developers in the world.

Contact us

Potix Corporation

info@zkoss.org
www.zkoss.org