Security Issues in ZKoss (ZUL) Applications: Identification and Mitigation

Introduction

ZKoss (ZK) is a rich Java-based web framework that uses **ZUL (ZK User Language)** files for UI definition. While ZK simplifies building UIs, developers must be vigilant about security. This report covers common vulnerabilities specific to ZUL files and advanced backend practices to secure ZK applications. We focus on identifying issues like script injection, cross-site scripting (XSS), and unsafe use of expressions, along with session/login security. Real-world ZUL examples, detection steps, and mitigation strategies are provided for each case.

Common Vulnerabilities in ZUL Files

1. Script Injection Vulnerabilities (`<zscript>` Blocks)

ZUL files allow embedding server-side scripts using the `<zscript>` element. By default, ZK uses Java for `<zscript>`, but it can run other languages (Groovy, JavaScript, Ruby, Python) via the `language` attribute

zkoss.org
. **Script injection** occurs when an application executes dynamically constructed code that includes untrusted input, leading to **remote code execution (RCE)**. Groovy scripts in `<zscript>` are particularly sensitive because Groovy is dynamic and powerful.
Example Vulnerable ZUL (Groovy Injection):

xml
CopyEdit
```
<window title="Admin Console">
  <zscript language="Groovy">
    // BAD: Evaluating user-provided script (highly insecure)
    def userScript = Executions.getCurrent().getParameter("code");
    if (userScript != null) {
        new GroovyShell().evaluate(userScript);  // Vulnerable:
executes arbitrary Groovy code
    }
  </zscript>
```

```
</window>
```

- In this example, an attacker who passes a malicious `code` parameter (e.g., `System.exit(0)` or spawning a shell) can execute arbitrary server-side code. Using `GroovyShell.evaluate()` on a dynamic string is a known code injection sink github.com

  docs.datadoghq.com
  .

- **Identification (Detection Steps):**
  - **Search ZUL files for `<zscript>` usage** – Review all occurrences of `<zscript>` and note the `language`. Pay extra attention to `language="Groovy"` or other scripting languages. These indicate places where server-side code is embedded.
  - **Examine for dynamic code execution** – Within each `<zscript>` block, look for use of classes like `GroovyShell`, `GroovyClassLoader`, `Interpreter`, or eval/execute methods. Also flag any use of `Executions.create**` methods that compile or run dynamic content. For Java `<zscript>`, check for use of reflection or `Executions.evaluate()` (if any).
  - **Trace input sources** – If the script constructs code from variables or parameters, determine if any of those come from user input (e.g., `Executions.getCurrent().getParameter`, component values, or session attributes set by users). Any concatenation of user data into code is a red flag.
  - **Real-world clues** – Often, script injection issues arise in admin or customization features (e.g., allowing admin to enter scripts for custom logic). Identify any such feature in documentation or code comments and review it thoroughly.
- **Mitigation Strategies:**
  - **Avoid Dynamic Evaluation:** The safest approach is not to evaluate user-supplied code at all. Implement required logic in Java (controllers or view models) rather than via dynamic Groovy scripts. This keeps code static and compiled, eliminating this injection vector.
  - **Restrict Scripting Capabilities:** If dynamic scripts are absolutely needed (e.g. plugin systems), do not expose them to untrusted users. Require authentication and limit this feature to admin roles. Additionally, sandbox the script environment:

disable dangerous classes/methods. (For example, configure a custom `ClassLoader` or use Groovy's secure features to whitelist permitted operations.)

- **Input Validation:** Rigorously validate or sanitize any input that might be used in script. *Never* pass raw user input to `GroovyShell.evaluate()` or similar. If you must accept expressions (say, a math formula), parse and validate the format before evaluation.

- **Use FindBugs/CodeQL rules:** Employ static analysis tools which have detectors for Groovy script injection. For instance, FindSecBugs has rules to catch usage of GroovyShell w
  github.com
  nput【4†L205-L212】. This can help flag vulnerable code during development.

- **Least Privilege:** Run the application with minimal privileges. Even if code injection occurs, a locked-down security manager or low-privileged OS user can limit damage. (Advanced setups might remove the Groovy engine entirely from production if not used.)

By treating any `<zscript>` as potential executable code and avoiding inclusion of unvalidated data, you prevent attackers from injecting scripts that the server would run.

2. Cross-Site Scripting (XSS) in ZUL Content

Cross-site scripting is a pervasive web vulnerability where malicious scripts are injected into pages v

zkoss.org
hers【1†L56-L64】. In ZK, XSS typically arises when **user input is included in ZUL views without proper encoding**. The ZK framework encodes many outputs by default (to help developers), but there are exceptions where developers must be cautious.
How ZK Handles Encoding: ZK automatically escapes special characters (`<`, `>`, `&`, etc.) for all standard input components and label outputs. For example, using a value binding in a textbox or label is safe:

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

- Even if `any_value` contains `<script>alert('XSS')</script>`, ZK will escape it (the `<`
  zkoss.org
  lt;`, etc.)【1†L68-L76】. In fact, an EL expression in a ZUL page implicitly renders a L

zkoss.org
is encoded【1†L81-L84】. **This means normal data binding to UI components is generally safe from XSS by default.**

- **Unsafe Cases (When ZK Does *Not* Encode):** Certain ZK components and techniques deliberately allow raw HTML for flexibility. These will **not auto-escape** content and thus can introduce XSS if misused:

`<html>` Component (Html.setContent) – Intended to embed raw HTML.
zkoss.org
encoded【29†L127-L136】. Example:

xml
CopyEdit
```
<html>${userComment}</html>
```

  - If `userComment` includes a script tag, it will be rendered as-is, executing malicious script. (Note: Since ZK 10.0.0, the Html component attempts to sanitize content by default to reduce XSS risk, but developers are advised *not* to rely on this and never include raw `<script>
zkoss.org
vided content【2†L103-L107】.)

  - **`<comboitem content="...">`** – Comboitem's content attribute is raw HTML
zkoss.org
ed【29†L127-L135】. Any user-supplied HTML here can execute scripts or HTML.

  - **Native HTML in ZUL (`<div xmlns="xhtml">` or `xmlns:native`)** – Embedding native HTML tags directly in ZU
zkoss.org
ZK's encoding【1†L87-L95】. E.g., `<div xmlns="native">${data}</div>` will inject raw data.

Client-side Utility Methods – Using `Clients.showNotification(String)`, `Clients.alert(String)`, or similar to display messages, and `Clients.evalJavaScript(String)` to run scri
zkoss.org
not encode their input by design【2†L113-L120】. They allow formatting or direct script execution on the client. If you pass user data directly, it can inject scripts. For example:

java

CopyEdit
```
// In a ZUL event listener or ViewModel:
String name = userService.getName();            // could be
attacker-controlled
Clients.evalJavaScript("showWelcome('" + name + "')");
```

  - If `name` contains a quote or `</script>`, it could break out of the string and run arbitrary JS. Similarly, `Clients.showNotification("Success: " + userInpu&#8203;:contentReference[oaicite:12]{index=12}er any HTML in` userInput`【2†L113-L120】. Developers must manually escape or sanitize in these cases.

  - **Page Directive Attributes** – All attributes in `<?page?>` (the ZUL pa
    zkoss.org
    are output without encoding【2†L129-L137】. In practice, if any page attribute is set from user data (rare, usually static design-time values like title), it could be an injection vector.

Example Vulnerable ZUL (XSS):

xml
CopyEdit
```
<window title="Feedback">
  <!-- Developer wants to allow some HTML in userComment -->
  <html>${param.userComment}</html>
  <button label="Preview"
onClick='Clients.showNotification(param.userComment, "info")'/>
</window>
```

- Here, if `param.userComment` is `<img src=x onerror="alert('XSS')">`, the `<html>` component will inject an actual image tag with the onerror script, executing the alert. The `Clients.showNotification` will also display it un
  zkoss.org
  ich could pop the alert or distort the UI【2†L113-L120】.

- **Identification (Detection Steps):**

  - **Find raw HTML uses** – Grep for `<html>` components or `.setContent(` calls in Java. These are likely points where raw output is used. Check if the content comes from user input (e.g., form fields, database content that users can influence).

    - **Check comboitem and others** – Search for `comboitem content=` in ZULs. Also look for usage of native namespaces (`xmlns:native` or `xmlns:xhtml`) which signal direct HTML embedding.

    - *Scan for Clients. calls** – In ZUL onClick/ onOK handlers or controllers, find calls to `Clients.evalJavaScript`, `Clients.showNotification`, `Clients.alert`, etc. **Audit what data is being passed**. If any parameter in those calls originates from a user (request param, component value, etc.) and isn't encoded, that's a potential XSS.

    - **Identify disabled encoding** – Although ZK encodes labels by default, ensure no code explicitly disables encoding. For instance, in older ZK, `Label.setRawValue()` or `disableClientEscaping` flags might output raw HTML. Such patterns should be treated as sinks for XSS.

    - **Pen-test dynamic content** – Run the app and input typical XSS payloads (e.g., `<script>alert(1)</script>` or `<img onerror=alert(1) src=nonexistent>` in form fields) and see if they are rendered in any response/UI without being neutralized. This dynamic testing complements code review to catch any missed spots.

- **Mitigation Strategies:**

    - **Prefer Safe Components**: Use ZK's normal components (Label, T
      zkoss.org
      for user-generated content, as they encode dangerous characters by default【29†L117-L125】. For example, display user comments in a `<label>` or `<textbox readonly="true">` rather than an `<html>` element. If you simply need multiline text display, ZK's Label supports multiline (or use `<label multiline="true">` which still escapes HTML).

    - **Sanitize if Raw HTML Needed**: If your use-case requires allowing some HTML (e.g., formatting in user comments), sanitize it on the server before rendering. Use libraries like **OWASP**
      zkoss.org
      r, **Jsoup**, or ZK's own utility `XMLs.escapeXML(String)`【2†L133-L140】 to strip or encode disallowed tags. *Example:* before setting content on an `<html>` component, do:
      `htmlComp.setContent(XMLs.escapeXML(userComment));` – this will

convert `<` to `&lt;`, neutralizing any tags.

- **No Scripts in HTML Content**: Neve
  zkoss.org
  ipt>`tags via user input. Even if ZK 10+ auto-sanitizes`<html>`content 【2†L103-L107】, it's wise to explicitly remove scripts or event attributes to be sure. **Tip:** If you must allow limited HTML (e.g.,`<b>`, `<i>`), whitelist those and strip everything else.

Escape Data in Client Calls: For `Clients.evalJavaScript`, don't inline user strings into JavaScript code. If you need to pass user data to a JS function, use JSON encoding or base64. For example:

java
CopyEdit
```
String safeJson = org.zkoss.json.JSONValue.escape(userInput);
Clients.evalJavaScript("showMessage(JSON.parse('" + safeJson + "'))");
```

- Better, use ZK's `JavaScriptValue` (if available) or send the data to client as an argument rather than constructing JS. For notifications, if you want to display raw text,
  zkoss.org
  howNotification(Strings.escape(userInput))`or supply the`raw=true` parameter appropriately after sanitizing.

- **Review ZK Encoding Gaps**: Be aware of which propertie
  zkoss.org
  ded【29†L126-L134】. For instance, avoid binding user data to `Html.setContent` or `Comboitem.setContent` unless sanitized. ZK's documentation explicitly warns these are not encoded and need developer care【29†L129-L137】. Incorporate these guidelines in code reviews.

- **Content Security Policy (CSP)** (Advanced): Consider using a strict CSP in your HTTP response headers to mitigate impact of any XSS that slips through. For example, disallow inline scripts except those from your domain. This is a second line of defense.

By encoding all user-supplied content or sanitizing in cases where raw HTML is needed, you can effectively prevent XSS in ZK applications. Remember that ZK's default is safe for most UI elements – problems usually arise when deliberately bypassing those safeties for flexibility.

3. Improper Use of EL Expressions and Dynamic Content Rendering

ZK supports powerful expression language (EL) binding and dynamic UI composition. **Improper use of these features can introduce security issues** such as unauthorized page inclusion, template injection, or expression evaluation risks. This category covers scenarios outside of simple XSS – for example, including the wrong ZUL page based on user input, or using EL in a way that leaks data or calls unsafe methods.

Unsafe Dynamic Page Inclusion (ZUL Path Injection): Developers sometimes compose pages dynamically. For instance, using a parameter to decide which ZUL to include:

xml
CopyEdit
```
<!-- Vulnerable if 'page' param is not validated -->
<include src="/views/${param.page}.zul"/>
```

- If `param.page` is attacker-controlled, this can load arbitrary Zul files. An attacker might access pages they shouldn't (e.g., `admin` or `../WEB-INF/config`). While the servlet spec prevents loading files from WEB-INF via direct URL, an include might still pull it if not protected (depending on ZK implementation). Even if not, an attacker could try values that cause unintended behavior or errors (possibly revealing stack traces or sensitive info). In general, **treat any user input that constructs a file path as dangerous** (similar to directory traversal vulnerabilities).

Template Injection via `Executions.createComponents`: ZK's API allows creation of components from ZUL content at runtime:

java
CopyEdit
```
String zulPath = Executions.getCurrent().getParameter("zulPage");
Executions.createComponents(zulPath, parentComponent, null);
```

- If `zulPage` is user-supplied (query param or form), an attacker could input the path to an admin page or a malicious Zul file on the server. Even worse, if the app ever takes raw ZUL markup from a user and calls `createComponentsDirectly(String zulMarkup, ...)`, that's effectively evaluating user code (similar to an HTML template injection leading to XSS or RCE depending on content). This is a serious issue – it's like allowing an attacker to feed your server a new ZUL page to render.

Improper EL Usage: ZK's EL in bindings is mostly safe because it doesn't eval arbitrary expressions from the client – the developer defines the expression in the Zul. However, *if* developers abuse it, e.g., by constructing EL strings dynamically or using it to call static methods unsafely, it can bite. For example, if a binding calls a static method that should be restricted:

xml
CopyEdit
```
<label value="${systemProperties['os.name']}"/>
```

- This might expose system info. Or using ZK's older scripting, one might do something like `${Class.forName(param.classname)}` which could trigger classloading. Such uses are uncommon but illustrate that EL can access server-side resources if not careful. The key point is **never feed untrusted input into EL evaluation**. ZK doesn't expose an eval EL function by default, so this is more about design: ensure your ZUL bindings don't inadvertently expose sensitive data or operations.

Example Issue (Dynamic Include):

xml
CopyEdit
```
<window apply="com.app.MyComposer">
  <!-- Show different page based on 'view' request parameter -->
  <zscript>
     String page = Executions.getCurrent().getParameter("view");
     if(page != null) {
         Include include = new Include("/zul/" + page + ".zul");
         self.appendChild(include);
     }
  </zscript>
</window>
```

- If an attacker accesses `?view=../../WEB-INF/sensitive`, this code will try to load `/zul/../../WEB-INF/sensitive.zul`. This might throw an error or potentially load a forbidden file if traversal is not blocked. Even if not, it allows scanning for valid ZUL pages or loading pages out of intended flow (e.g., loading an admin.zul without going through login). This is an **access control issue** as well as a possible injection issue (if a crafted path breaks out of the expected directory).

- **Identification (Detection Steps):**
  - **Audit includes and navigation logic** – Search for `<include src="${...}">` in ZUL or any usage of `Include` components in code. Also search Java code for `Executions.createComponents`,

`Executions.createComponentsDirectly`, `Executions.include` or similar. Wherever a path is constructed dynamically, flag it.

  - **Trace input for dynamic paths** – If a dynamic include uses a variable, find where that variable comes from. Is it a request parameter, session attribute, or a hardcoded mapping? If it's user-controlled and not validated against an allow-list of safe values, it's a potential vulnerability.

  - **Look for eval-like patterns** – Although ZK's binder doesn't provide an eval from string, check if the code uses any evaluation utilities (perhaps using `javax.el.ELProcessor` or OGNL). Also, check `<zscript>` blocks for usage of `Executions.evaluate()`, `Path.getComponent()` with dynamic expressions, or custom EL resolvers that might be abused.

  - **Examine data-binding expressions** – Review your MVVM bindings (in `viewModel` and data bindings) for any suspicious expressions that might expose more than intended. E.g., using `${vm.someMap[param]}` where `param` could be user input – ensure it can't be abused to get arbitrary map keys. Or if using **ZK Functions** or static method calls in EL, ensure they can't do something unsafe. It's rare to find critical issues here, but a quick check can reveal oversights.

  - **Test scenario** – Try manipulating any request parameters that seem to select pages or templates. If you find pages that aren't meant to be user-facing can be loaded by direct URL or parameter, that's a broken access control. Ensure unauthorized users cannot reach admin or internal ZULs by guessing paths.

- **Mitigation Strategies:**

Validate and Whitelist: Never use raw user input to select a ZUL page or template. If you need to switch views based on input, maintain a **whitelist map** of allowed page names. For example:

java
CopyEdit
```
Map<String,String> pageMap = Map.of(
    "home","/zul/home.zul",
    "dashboard","/zul/dashboard.zul"
);
String page = param("view");
if(pageMap.containsKey(page)) {
   Include include = new Include(pageMap.get(page));
```

```
    // ...
} else {
    // invalid page requested
}
```

- This ensures only known good pages load. Any attempt to brea
  zkoss.org
  `../`) will simply not match the whitelist.

- **Keep ZUL files out of public reach**: Place sensitive ZUL files (like admin pages) under `WEB-INF` or configure your web.xml to forbid direct access. While ZK pages are typically served through the ZK servlet and not directly as files, it's wise to treat them as server-side view templates. If a page should only load after login, do not allow it to be included without proper checks.

- **Use Executions.createComponents safely**: If you use `createComponents` with a path, follow the whitelist approach above. If using `createComponentsDirectly` with a template string, avoid using user-provided template data entirely. That method is meant for cases like generating a small snippet of components – feed it only constant or server-generated strings. *Do not pass user text into it.* If you must allow users to define templates (e.g., custom form designer), heavily sanitize that input and strip any `<zscript>` or dangerous elements from it before evaluation.

- **Avoid Overly Dynamic EL**: In data binding expressions, stick to property accesses and method calls you control. Avoid exposing general maps or scripting objects that could be misused. If you use something like ZK's FunctionMapper to call static methods, restrict it to safe utilities. And absolutely do not design a system where end-users input EL expressions – ZK has no such feature by default, so don't add one without sandboxing.

- **Framework Patches**: Stay updated on ZK security patches. For instance, ZK 10 introduced stricter handling of GET/POST to prevent misuse of the servlet【10†L93-L96
  zkoss.org
  ed `InaccessibleWidgetBlockService` by default to block client-altered component access【10†L69-L77】. Use the latest ZK version so that known vulnerabilities in dynamic content handling (e.g., old file upload issues, path traversal bugs)
  nvd.nist.gov
  or example, a recent CVE-2022-36537 was patched to prevent an attacker from accessing files via the AuUploader component by crafting requests【5†L123-L131】【5†L123-L125】. Keeping up-to-date shields you from such framework-level

flaws.

By reducing overly dynamic behavior and tightly controlling what content and pages can be rendered, you eliminate an entire class of injection and inclusion problems. Essentially, **never let the user decide what code or page executes** – always mediate and validate their requests.

Backend Java Security Practices for ZK Applications (Intermediate to Advanced)

While the focus is on ZUL files, the backend Java code in a ZK app plays a critical role in security. Here we outline some practices (beyond the basics) that experienced developers should follow:

- **Server-side In**
  [zkoss.org]
  on & Injection Prevention: ZK is mostly a UI framework; it won't automatically secure your database or backend integrations【12†L66-L74】. Apply standard injection prevention in your Java code:
    - Us
      [zkoss.org]
      rized queries or ORM protections** for database access (prevents SQL injection). Never concatenate user input into SQL directly【12†L69-L72】.
    - For LDAP queries, OS commands, or other interpreters, similarly ensure inputs are sanitized or use safe APIs.
    - Validate inputs rigorously using whitelist patterns. ZK provides a `Constraint` mechanism on input components for basic validation, but also enforce checks on the server side (e.g., using Jakarta Bean Validation or manual checks) to ensure malicious data is rejected.
    - Beware of **insecure deserialization** if you use Java serialization (e.g., storing user-provided objects in session or cache). This is more a concern if you accept serialized data or use libraries that might do so. Use modern serialization alternatives (JSON, etc.) and never deserialize unknown data.
- **Sensitive Data Handling:** Only send to the UI what is necessary for the user. Because ZK keeps components and data in server memory, you might be tempted to attach lots of data to components. Remember:

    - **Do not store secrets in component attributes** that end up sent to client. For example, avoid `<label value="${user.password}"/>` – you wouldn't do that plainly, but even storing it in a non-visible attribute could be risky if the client can somehow see it. Keep sensitive info server-side (in session or security context).

    - If you have to display partial sensitive data (like email or ID), mask it (e.g., show only part of it).

    - Never expose internal file paths, config values, or stack traces in the UI. Catch exceptions and show generic errors.

    - Store files (like exports or reports) in non-web accessible locations (e.g., under WEB-INF or outside web root)【12†L79-L84】, and stream them to the user after permission checks rather than linking directly.

- **Access Control (Enforce on Server):** ZK's UI controls (like disabling a button or hiding a component) are not sufficient for security. Always check permissions on the server before performing an action:

    - Use a robust authentication and authorization system. ZK itself doesn't provide login/auth – integrate with **Spring Security** or container security【12†L75-L79】. These frameworks can handle session management, password hashing, and URL/page access rules. ZK pages can be protected via filters or configured to require certain roles.

    - On every sensitive server operation (event lis
      zkoss.org
      mand), verify the user's role/rights. For example, if a button deletes a record, the event handler on the server should ensure the current user is allowed to delete that record, even if the UI hid the button for others. This protects against forged requests or clients manipulating the UI.

    - Use ZK's `InaccessibleWidgetBlockService` (enabled by default since 10.0.0) which **blocks events from components that are not supposed to be usable (like disabled or invisible ones)**【10†L69-L77】. This is a nice safety net: if an attacker enables a hidden admin button via browser dev tools, ZK will ignore the click on the server. Earlier versions of ZK didn't have this on by default; if you use an older version, consider e
      zkoss.org
      anually via a listener in `web.xml` as documented by ZK. This service complements your own access checks.

    - **Broken Access Control check**: Ensure that pages that require login cannot be loaded without login. For instance, if using a custom login, each ZUL page's controller (composer or viewModel) can redirect unauthenticated users to the login page in its initialization if needed. Or use a servlet filter to intercept requests to `/secure/*` ZULs. This prevents direct access by guessing URLs【26†L61-L69】.

- **Session Management:** Proper session security keeps user sessions from being hijacked or misused:

Session Fixation: When a user logs in, invalidate any old session and issue a new one. If you use Spring Security, this is handled for you (it by default creates a new session on authentication)【25†L1-L4】. If not, manually do:

java
CopyEdit
```
HttpSession oldSession = request.getSession(false);
if(oldSession != null) { oldSession.invalidate(); }
HttpSession newSession = request.getSession(true);
// copy necessary attrs to newSession if needed
```

    - This ensures an attacker cannot use an existing session ID (from before login) to hijack the session.
    - **Secure Cookies:** Configure your session cookie with `HttpOnly` and `Secure` flags (usually in web.xml or container settings). HttpOnly prevents client-side scripts from reading the cookie, and Secure ensures it's only sent over HTTPS. This is critical to mitigating XSS (stops it from stealing session cookies) and safeguarding against network sniffing.
    - **Session Timeout:** Set a reasonable session timeout in web.xml (or programmatically). Inactivity timeouts reduce the window for an attacker to reuse a stolen session. For high-security apps, consider an absolute timeout (max session lifespan) or implement re-authentication for critical actions.
    - **Concurrent Session Control:** Depending on requirements, you might limit users to one session at a time or log out previous sessions on new login, etc., to reduce session abuse. Spring Security can do this; otherwise manage session IDs in a server-side store.
    - **Session Storage:** Store minimal data in the session. Keep only identifiers and necessary info (e.g., user ID, roles). Avoid placing large objects or sensitive data zkoss.org

it increases risk (and memory usage). Also, if using session clustering, test that serialized session data doesn't include anything sensitive unexpectedly.

- **Login Security:** Ensure that the authentication process itself is secure:
  - **Hash Passwords:** On the server side, never store plaintext passwords. Use strong hashing (e.g., bcrypt) with salt for any password storage. On login, compare hashed values. This is outside ZK's scope but crucial if you implement your own auth.
  - **Transport Security:** Always use HTTPS for login pages and any page with sensitive data. ZK apps often are behind a TLS termination, but double-check that credentials aren't sent over plain HTTP. If using ZK's AJAX for login (e.g., submitting a form via an event), ensure the page is loaded on HTTPS and check the network calls.
  - **Brute-force protection:** Implement account lockou
    zkoss.org
    tial backoff on failed logins. ZK doesn't do this inherently, but you can track in your auth service. For
    zkoss.org

    zkoss.org
    ttempts, lock the account for a period.
  - **Multi-factor Authentication:** For advanced security, integrate MFA for login if the application warrants it. ZK can integrate with whatever solution (since the backend is Java).
  - **Logout and Session Invalidation:** Provide a logout function that truly invalidates the session (use `Sessions.getCurrent().invalidate()` or `sessi&#8203;:contentReference[oaicite:31]{index=31}e()`). Also consider the **"logout everywhere"** scenario if user requests (invalidate all sessions of that user).
- **Other Best Practices:**
  - **CSRF Protection:** ZK's architecture makes CSRF attacks more difficult by using unique component/desktop IDs in every
    zkoss.org
    89-L98】【16†L109-L117】. An attacker would have to guess a moving target ID, which is impractical. However, if your app has any non-AJAX endpoints (e.g., file upload servlets, REST endpoints), protect those with CSRF tokens. ZK itself suggests using the **Synchronizer Token Pattern** with a hidden token in forms if

needed【16†L109-L117】. Ensure any custom AJAX endpoints or custom servlets in your app also check origin or require a token.

- **Clickjacking Prote
  stackoverflow.com
  et `X-Frame-Options: DENY` or `SAMEORIGIN` on responses (or CSP frame-ancestors) to prevent your ZK pages from being iframed by malicious sites. This isn't ZK-specific but important if your app has sensitive UI operations.

- **Security Headers:** In addition to CSP and X-Frame-Options, consider `X-XSS-Protection` (for older IE), `X-Content-Type-Options: nosniff`, etc., via a servlet filter. These just add extra protection layers in browsers.

- **Logging and Monitoring:** Implement robust logging for security-related events: log logins (and failures), important user actions (especially admin actions), and suspicious inputs (e.g., catches of exceptions that indicate attempted attacks). Monitor these logs. Set up alerts for multiple failed logins or unusual behavior. This helps detect if someone is probing your ZK application for vulnerabilities.

- **Use Updated Components/Libraries:** Keep the ZK framework and any add-ons up to date. As noted, using the latest ZK version protects you from known issues (such as the file upload CVE). Also update dependent libraries (e.g., if you use Commons FileUpload, etc.). Outdated components are one of OWASP Top risks【10†L46-L53】.

- **Penetration Testing:** Periodically have security testing done on the application. Tools can static-scan your code for known vulnerable patterns (including the ZUL files) and dynamic scanners can try XSS, injection, etc., in running applications. Given ZK's rich interface, some automated scanners might not navigate it fully, so consider manual pen-testing focusing on the points discussed: try injecting scripts in form inputs, attempt to bypass UI restrictions, force error messages, and see if any sensitive data leaks.

Session and Login Security in ZKoss Applications

(Session management was partly covered above, but here we emphasize specifically within the context of ZK.)

ZK applications are stateful by nature – user interactions are managed via server-side sessions and the framework's **Desktop** (which is like a UI session token). As such, session and authentication security are critical.

Session Security in ZK

ZK leverages the Java EE session (`HttpSession`) to store component state and user data. Each browser tab or window corresponds to a ZK **desktop** identified by a unique ID, which ZK uses to tie AJAX requests to the correct session and page【16†L89-L98】.

- **CSRF Mitigation via Desktop ID:** Each ZK page load generates a unique **desktop ID** token that is automatically sent with every AJAX request【16†L109-L117】【16†L89-L98】. This functions similar to a CSRF token – if an attacker tries to forge an AJAX request, they would need this desktop ID, which is hard to guess. ZK also checks server-side that the component IDs in requests match those orig
  zkoss.org
  ered, preventing an attacker from invoking actions on components that weren't present or have been removed【16†L89-L98】. This means out of the box, ZK has some CSRF resistance. Nonetheless, **do not become complacent**: if you have endpoints outside ZK's AJAX (file upload, etc.), protect them as mentioned earlier. And you can reinforce security by also checking the `Origin/Referer` on incoming requests【16†L72-L80】 or adding your own CSRF token in forms if needed (especially for traditional forms that might post outside the ZK event mechanism).

- **Session Hijacking and Hardening:**

  - As discussed, always invalidate the session on login to prevent fixation. If using ZK Spring Security, this is handled for you【25†L1-L4】.

  - Make sure JSESSIONID cookie is marked HttpOnly and Secure. This is typically done in your container or `web.xml` (`<cookie-config>`).

  - ZK does not transmit the session ID in URLs by default (it uses cookies unless configured otherwise). Avoid URL rewriting (disable `URLSessionTracking` unless absolutely needed) to not expose session IDs.

  - If your app is high-value, consider implementing detection of session hijacking – e.g., if one session ID suddenly comes from a new IP or user agent, you might invalidate it. This can be complex (and can hit legitimate cases like mobile network switching IPs), so implement carefully if at all.

  - Educate users to log out and close the browser, especially on shared computers, to destroy the ZK desktop and session.

- **ZK Specific Session Tips:**

  - **Max Desktops/Requests**: ZK provides configurations to limit how many desktops (tabs) a session can create and how many requests per session to prevent DoS【29†L141-L149】【29†L135-L139】. In `zk.xml` or `WEB-INF/zk.xml` config, you can set `<max-desktops-per-session>` and `<max-requests-per-session>` to sane values to mitigate abuse (e.g., a malicious script opening hundreds of tabs or flooding requests). Consider adjusting these if your app is susceptible to such misuse.

- **Server Push**: If you use server-push or WebSockets, e
  hawkchen.gitbooks.io

  hawkchen.gitbooks.io
  sent to clients are properly authenticated/authorized on the server side. Don't broadcast sensitive data to sessions that shouldn't receive it. ZK's event scope should handle this, but double-check your push event code.

Login Security in ZK

Since ZK delegates authentication to the developer, secure login flows are crucial:

- **Protecting the Login Page**: If the login is a ZUL page with a form (username/password fields), deliver that page over HTTPS. Use ZK's features (like `<textbox type="password">`) to ensure the password input is masked on UI. There's no built-in encryption of the password on submit (that's what HTTPS is for), so standard web best practices apply.

- **Using Spring Security or JAAS**: Instead of reinventing login, prefer using **Spring Security integration with ZK**【23†L8-L12】. ZK has documentation on integrating it【23†L8-L12】, which allows you to use all of Spring Security's features (BCrypt password encoding, remember-me tokens, method security, etc.). This also makes things like session fixation protection, CSRF token on non-AJAX forms, and authentication flows much easier to manage.

- **Custom Login**: If implementing manually, as shown in many ZK examples, do not follow them blindly when it comes to password handling. For instance, some tutorial code compares plaintext passwords directly【26†L99-L107】 – you should improve that by hashing the stored password. Also, add a delay or captcha after successive failed attempts to thwart brute force.

- **Session Management Post-Login**: After successful login, besides session renewal, you might want to store an object like `UserCredential` in the session (as in ZK Essentials examples【26†L75-L83】【26†L106-L114】). Make sure this object only contains necessary info (user id, roles, name). Avoid storing the password or other sensitive tokens in session. This session attribute can be checked in each page or composer to verify the user is logged in.

Page Access Checks: Implement a simple check in page initializations: e.g., in each secured ZUL's controller:

java
CopyEdit
```
if (Sessions.getCurrent(false) == null ||
Sessions.getCurrent().getAttribute("userCredential") == null) {
    Executions.sendRedirect("/login.zul");
    return; // stop further init
}
```

- This ensures that if someone tries to load the page without a login, they get redirected. It's a backstop if someone found a direct URL. In a larger app, a filter or a base composer class that all secure pages extend can do this globally.

- **Logout**: Provide a logout button that calls `Sessions.getCurrent().invalidate()` or uses Spring Security's logout if integrated. After invalidation, ZK will detach the desktop. Redirect the user to a public page (login or home). Also, clear any client-side tokens or identifiers if you set any.

- **Audit Login Events**: As mentioned, log successful and failed logins (but not
 zkoss.org

  zkoss.org
  se logs for suspicious activity.

Security Tips for ZK Components and Data Exposure

Finally, here are additional tips for safe usage of specific ZK components and patterns that often come up in advanced ZK development:

- **ZK Components Executing Scripts**: Use them carefully.

  - The `<zscript>` component is powerful but consider alternative designs. For maintainability and security, it's often better to put Java logic in a **Composer (MVC)** or **ViewModel (MVVM)** class, rather than inline in ZUL. This way, your code is in Java (statically compiled, checked) and less prone to injection mistakes. `<zscript>` is best used for very small glue logic or legacy code. If you do use `<zscript>`, restrict it to Java language (the default) when possible, and avoid dynamic evals as described earlier.

    - If you need client-side scripting, ZK offers `<client>` annotations or `<attribute>` with `client*` to attach client-side listeners, etc. Those are safer than constructing raw script. Use ZK's **JavaScript APIs** or Angular-style data binding rather than injecting script strings. For example, use `onClick="someFunction()"` that's declared in a script file, rather than building a script with user content.
    - **Clients.evalJavaScript** – we reiterate: this should be a last resort. Before using it, ask if ZK has a built-in way to achieve the effect. Often, ZK components/attributes or the `AuResponse` mechanism can do complex tasks without manual JS. Minimizing direct JS reduces XSS risk.
- **Components that Expose Data**:
    - ZK abstracts the client-server communication, but remember that certain data must go to the client to render the UI. For example, if you have a listbox with 100 items, those items' data might be sent to the browser (unless you use paging or on-demand loading). If those items contain sensitive info that the user shouldn't have yet, don't load them all. Load only what's needed for display, or mask certain fields until the user takes an action that is authorized.
    - **ListModel and Model objects**: If you bind a component to a model (say a `ListModelList`), all data in that model could be exposed to the

Security Issues in ZKoss (ZUL) Applications: Identification and Mitigation

Introduction

ZKoss (ZK) is a rich Java-based web framework that uses **ZUL (ZK User Language)** files for UI definition. While ZK simplifies building UIs, developers must be vigilant about security. This report covers common vulnerabilities specific to ZUL files and advanced backend practices to secure ZK applications. We focus on identifying issues like script injection, cross-site scripting (XSS), and unsafe use of expressions, along with session/login security. Real-world ZUL examples, detection steps, and mitigation strategies are provided for each case.

Common Vulnerabilities in ZUL Files

1. Script Injection Vulnerabilities (`<zscript>` Blocks)

ZUL files allow embedding server-side scripts using the `<zscript>&#8203;:contentReference[oaicite:39]{index=39}default, ZK uses Java for` <zscript>`, but it can run other languages (Groovy, JavaScript, Ruby, Python) via the` language`attribute【8†L49-L57】. **Script injection** occurs when an application executes dynamically constructed code that includes untrusted input, leading to **remote code execution (RCE)**. Groovy scripts in`<zscript>` are particularly sensitive because Groovy is dynamic and powerful.

Example Vulnerable ZUL (Groovy Injection):

xml
CopyEdit
```
<window title="Admin Console">
  <zscript language="Groovy">
    // BAD: Evaluating user-provided script (highly insecure)
    def userScript = Executions.getCurrent().getParameter("code");
    if (userScript != null) {
        new GroovyShell().evaluate(userScript);  // Vulnerable:
executes arbitrary Groovy code
    }
  </zscript>
</window>
```

- In
  zkoss.org

  zkoss.org
  er who passes a malicious `code` parameter (e.g., `System.exit(0)` or spawning a shell) can execute arbitrary server-side code. Using `GroovyShell.evaluate()` on a dynamic string is a known code injection sink【4†L205-L212】【22†L3204-L3212】.

- **Identification (Detection Steps):**

  - **Search ZUL files for `<zscript>` usage** – Review all occurrences of `<zscript>` and note the `language`. Pay extra attention to `language="Groovy"` or other scripting languages. These indicate places where server-side code is embedded.

  - **Examine for dynamic code execution** – Within each `<zscript>` block, look for use of classes like `GroovyShell`, `GroovyClassLoader`, or eval/execute methods. Also flag any use of `Executions.create*` methods that compile or run dynamic content. For Java `<zscript>`, check for use of reflection or `Executions.evaluate()` (if any).

  - **Trace input sources** – If the script constructs code from variables or parameters, determine if any of those come from user input (e.g., `Executions.getCurrent().getParameter`, component values, or session attributes set by users). Any concatenation of user data into code is a red flag.

  - **Real-world clues** – Often, script injection issues arise in admin or customization features (e.g., allowing admins to input scripts for custom logic). Identify any such feature in documentation or code comments and review it thoroughly.

- **Mitigation Strategies:**

  - **Avoid Dynamic Evaluation:** The safest approach is not to evaluate user-supplied code at all. Implement required logic in Java (controllers or view models) rather than via dynamic Groovy scripts. This keeps code static and compiled, eliminating this injection vector.

  - **Restrict Scripting Capabilities:** If dynamic scripts are absolutely needed (e.g. plugin systems), do not expose them to untrusted users. Require strong authentication and limit this feature to admin roles. Additionally, sandbox the script environment: disable dangerous classes/methods. For example, configure a custom Groovy `ClassLoader` or use Groovy's secure features to whitelist permitted operations.

  - **Input Validation:** Rigorously validate or sanitize any input that might be used in a script. *Never* pass raw user input to `GroovyShell.evaluate()` or similar. If you must accept expressions (say, a math formula), parse and validate the format before evaluation.

  - **Use Static Analysis Tools:** Employ tools which have detectors for Groovy script injection. For instance, FindSecBugs has rules to catch usage of GroovyShell with dynamic input【4†L205-L212】. This can help flag vulnerable code during development.

  - **Least Privilege:** Run the application with minimal privileges. Even if code injection occurs, a locked-down security manager or low-privileged OS account can limit damage. (Advanced setups might remove scripting engine libraries entirely from production if not used.)

By treating any `<zscript>` as executable code and avoiding inclusion of unvalidated data, you prevent attackers from injecting scripts that the server would run.

2. Cross-Site Scripting (XSS) in ZUL Content

Cross-site scripting is a pervasive web vulnerability where malicious scripts are injected into pages viewed by others【1†L56-L64】. In ZK, XSS typically arises when **user input is included in ZUL views without proper encoding**. The ZK framework encodes many outputs by default (to help developers), but there are exceptions where you must be cautious.

How ZK Handles Encoding: ZK automatically escapes special characters (`<`, `>`, `&`, etc.) for all standard input components and label outputs. For example, using a value binding in a textbox or label is safe:

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

- Even if `any_value` contains `<script>alert('XSS')</script>`, ZK will escape it (the `<` becomes `&lt;`, etc.)【1†L68-L76】. In fact, an EL expression in a ZUL page implicitly renders a Label, which is encoded【1†L81-L84】. **This means normal data binding to UI components is generally safe from XSS by default.**
- **Unsafe Cases (When ZK Does *Not* Encode):** Certain ZK components and techniques deliberately allow raw HTML for flexibility. These will **not auto-escape** content and thus can introduce XSS if misused:

`<html>` Component (Html.setContent) – Intended to embed raw HTML. Content is not encoded【29†L127-L135】. Example:

xml
CopyEdit
```
<html>${userComment}</html>
```

  - If `userComment` includes a script tag, it will be rendered as-is, executing malicious script. (Note: Since ZK 10.0.0, the Html component sanitizes content by default to reduce XSS risk, but developers are advised *not* to rely solely on this and never include raw `<script>` in user-provided content【2†L103-L107】.)

- **Comboitem Content** – The `<comboitem>` component's `content` attribute is raw HTML and not encoded【29†L127-L135】. Any user-supplied HTML here can execute scripts.
- **Native HTML in ZUL** – Embedding native HTML tags directly in ZUL (using the `native` or `xhtml` namespace) bypasses ZK's encoding【1†L87-L95】. For example, `<div xmlns="native">${data}</div>` will inject raw HTML from `${data}`.

Client-side Utility Methods – Using `Clients.showNotification(String)`, `Clients.alert(String)`, or `Clients.evalJavaScript(String)`. These do **not** encode their input by design【2†L113-L120】. They allow formatting or direct script execution on the client. If you pass user data directly, it can inject scripts. For example:

java
CopyEdit
```
// In a ZUL event listener or ViewModel (Java code)
String name = userService.getName();            // could be
attacker-controlled
Clients.evalJavaScript("showWelcome('" + name + "')");
```

- If `name` contains a quote or `</script>`, it could break out of the string and run arbitrary JS. Similarly, `Clients.showNotification("Success: " + userInput)` will render any HTML in `userInput`【2†L113-L120】. Developers must manually escape or sanitize data in these cases.
- **Page Directive Attributes** – All attributes of the ZUL `<?page?>` directive are output without encoding【2†L129-L137】. In practice these are usually static (set by developers), but if any page attribute were set from user input, it could be a vector.

Example Vulnerable ZUL (XSS):

xml
CopyEdit
```
<window title="Feedback">
  <!-- Developer allows some HTML in userComment -->
  <html>${param.userComment}</html>
  <button label="Preview"
```

```
          onClick='Clients.showNotification(param.userComment,
"info")'/>
</window>
```

- If `param.userComment` is `"<img src=x onerror=alert(1)>"`, the `<html>` component will inject an actual image tag with an onerror script, triggering an alert. The `Clients.showNotification` will also display it unsanitized, which could execute the script or break the UI【2†L113-L120】.

- **Identification (Detection Steps):**

  - **Find raw HTML uses** – Grep for `<html>` components or `.setContent(` calls in Java. These are likely points where raw output is used. Check if the content comes from user input (e.g., form fields or database content that users can influence).

  - **Check comboitem and native tags** – Search for `comboitem content=` in ZULs. Also look for usage of `xmlns:native` or `xmlns:xhtml` in ZUL files which signals direct HTML embedding.

  - *Scan for Clients. calls** – In ZUL onClick/onChange handlers or controllers, find calls to `Clients.evalJavaScript`, `Clients.showNotification`, `Clients.alert`, etc. **Audit what data is being passed**. If any argument originates from a user (request param, component value, etc.) and isn't encoded or sanitized, that's a potential XSS sink.

  - **Identify disabled encoding** – Ensure no code explicitly disables encoding on components. E.g., calling `Label.setRawValue()` (if used) or setting a label's `encode=false` (in older ZK, Label had a flag to allow HTML). Such patterns should be treated as dangerous unless the content is trusted.

  - **Dynamic testing** – Run the app and input typical XSS payloads (e.g., `<script>alert(1)</script>` or `"><svg onload=alert(1)>` in form fields) to see if they appear unescaped in the UI. This helps catch any XSS issues that static review might miss.

- **Mitigation Strategies:**

  - **Prefer Safe Components:** Use ZK's normal components (Label, Textbox, etc.) for user-generated content, as they encode dangerous characters by default【29†L117-L125】. For example, display user comments in a `<label value="${comment}"/>` rather than an `<html>` element. If multiline text is needed, use `<label multiline="true">` (which still escapes HTML) or a readonly Textbox.

  - **Sanitize if Raw HTML Needed:** If your use-case requires allowing some HTML (e.g., bold/italic in user input), sanitize it on the server before rendering. Use libraries like **OWASP Java HTML Sanitizer** or **Jsoup** to whitelist tags and remove scripts. ZK's utility `XMLs.escapeXML(String)` can escape text【2†L133-L140】, though that will strip **all** HTML. For partial HTML, a sanitizer that allows certain tags is better. Always strip `<script>` tags or `on*` event attributes entirely.

  - **No Inline Scripts:** Never allow raw `<script>` tags via user input. Even with ZK 10+ auto-sanitizing `<html>` content, do not rely on it completely【2†L103-L107】 – explicitly remove or disallow script content.

Escape Data in Client Calls: For methods like `Clients.evalJavaScript`, avoid building JavaScript by string concatenation. If you need to pass user data to a client-side function, use JSON encoding or ZK's JSON utilities. For example:

java
CopyEdit
```
String safeName = org.zkoss.json.JSONValue.escape(name);
Clients.evalJavaScript("showWelcome('" + safeName + "')");
```

  - This ensures special characters in `name` are escaped in the JavaScript context. Similarly, for `showNotification`, if you want to display user text, use `Strings.escape(userInput)` (from ZK utils) or send it as a second parameter indicating content should be treated as text.

  - **Be Aware of Encoding Gaps:** Remember which components/properties do not auto-encode【29†L127-L135】. Avoid binding user data to them unless you sanitize first. If you use `Html.setContent()` or `<comboitem content=...>` for user data, you are responsible for cleaning it【29†L129-L137】. Incorporate these rules in code reviews or create a checklist for ZUL code.

  - **Content Security Policy (CSP):** (Advanced) Consider using a strict CSP header to mitigate the impact of any XSS that might occur. For example, disallow inline scripts and only allow scripts from your domain. This won't fix XSS, but it can make exploits harder (e.g., preventing injected scripts from executing or loading external resources).

By encoding or sanitizing all user-supplied content, you can prevent XSS in ZK applications. ZK's default behavior is safe for standard use; issues arise when developers bypass those safeties for flexibility, so weigh that need carefully against security risks.

3. Improper Use of EL Expressions and Dynamic Content Rendering

ZK supports powerful expression language (EL) binding and dynamic UI composition. **Improper use of these features can introduce security issues** such as unauthorized page inclusion, template injection, or data exposure. This category covers scenarios like including the wrong ZUL page based on user input and unsafe use of ZK's dynamic UI creation.

Unsafe Dynamic Page Inclusion (ZUL Path Injection): Developers sometimes compose pages dynamically. For instance, using a parameter to decide which ZUL to include:

xml
CopyEdit
```xml
<!-- Vulnerable if 'page' param is not validated -->
<include src="/views/${param.page}.zul"/>
```

- If `param.page` is attacker-controlled, this can load arbitrary ZUL files. An attacker might access pages they shouldn't (e.g., `admin.zul` or other users' pages). While direct access to `/WEB-INF` is blocked by the servlet container, including a page might still expose sensitive UIs or at least reveal their existence. **Treat any user input that influences file paths as dangerous** (akin to directory traversal vulnerabilities).

Template Injection via `Executions.createComponents`: ZK's API allows creation of components from ZUL content at runtime:

java
CopyEdit
```java
String zulPath = Executions.getCurrent().getParameter("zulPage");
Executions.createComponents(zulPath, parentComponent, null);
```

- If `zulPath` is user-supplied (from a query param, etc.), an attacker could try to load unintended zul files. Even more dangerous is `Executions.createComponentsDirectly(String zulMarkup, ...)` which evaluates a Zul markup string. Passing user input to that is equivalent to letting users execute code or HTML in your app. This is a serious risk – essentially remote template injection leading to XSS or worse.

- **Improper EL Usage:** ZK's EL in data binding is generally safe because it's defined by the developer, not the user. However, misuse can occur:
  - If you design something where user input is fed into an EL evaluation (e.g., letting users specify an EL expression for a field binding), this could be abused to call unintended methods. While ZK doesn't provide a straightforward way for users to supply EL, be cautious of any feature that might eval EL from strings.
  - Another aspect is accidentally exposing server data via EL. For example, ZK EL might allow accessing static resources or system properties if you bind them. `${systemProperties['os.name']}` would display the server OS name. That's not an "attack" per se, but leaking such info could aid an attacker. Review your ZUL bindings to ensure you're not exposing sensitive server information.
  - Ensure that data-binding expressions do not call methods with side effects unless intended. In MVVM, methods annotated with `@Command` or `@GlobalCommand` should include security checks as needed, since binding can trigger them based on UI actions.

Example Issue (Dynamic Include):

xml
CopyEdit
```
<window apply="com.app.MyComposer">
  <!-- Show different page based on 'view' request parameter -->
  <zscript>
     String page = Executions.getCurrent().getParameter("view");
     if(page != null) {
         Include inc = new Include("/zul/" + page + ".zul");
         self.appendChild(inc);
     }
  </zscript>
</window>
```

- If an attacker accesses `?view=adminDashboard`, this code will load `adminDashboard.zul` even if the user isn't an admin. Or `?view=../config` might attempt to load `../config.zul` (likely an invalid page, but it could expose an error or be manipulated to traverse directories). This is both an injection and an access control problem – the user can force the application to include pages out of the intended flow.

- **Identification (Detection Steps):**
    - **Audit includes and navigation logic** – Search for `<include src="${...}">` in ZUL or any usage of the `<include>` component with variable input. Also search Java code for `Executions.createComponents(` or `Include` object creation. Wherever a path is built dynamically, flag it.
    - **Trace inputs for dynamic paths** – If a dynamic include or component creation uses a variable, find how that variable is set. Is it from `Executions.getCurrent().getParameter()`? from a cookie? from user preferences? If it's user-controlled and not validated against an allowlist, it's a potential vulnerability.
    - **Check for direct ZUL eval** – Look for any use of `Executions.createComponentsDirectly` or similar functions that take raw ZUL content. These should never be fed with unsanitized external input. Also, verify that any custom UI templating you do (if any) doesn't allow injection.
    - **Review EL bindings** – Scan through ZUL files for unusual EL expressions. If you see something like accessing system properties, static classes, or executing expressions that could divulge data, note them. For instance, if a Label value is bound to something like `${vm.secretData}`, ensure that `secretData` is not something that should stay on the server. Usually, the presence of sensitive info in the view model implies it will go to the client. If it's not needed on the client, don't bind it.
    - **Test parameter manipulation** – Try altering any URL parameters or UI state that selects templates. If your app has a "page" or "view" parameter, attempt to load pages you shouldn't by changing it. If you find you can access an admin page while not logged in, that's a major issue to fix (via proper access control, see mitigations below).
- **Mitigation Strategies:**

Validate and Whitelist: Never use raw user input to select a ZUL page or template. If you need to switch views based on input, maintain a **whitelist** (mapping) of allowed pages. For example:

java
CopyEdit
```
Map<String, String> pageMap = new HashMap<>();
pageMap.put("home", "/zul/home.zul");
pageMap.put("dashboard", "/zul/dashboard.zul");
```

```
...
String view = Executions.getCurrent().getParameter("view");
if (pageMap.containsKey(view)) {
    Include inc = new Include(pageMap.get(view));
    parent.appendChild(inc);
} else {
    // Invalid page request - handle error or redirect
}
```

- This ensures only known pages load. Any attempt to supply a different value is ignored or handled as an error.

- **Secure ZUL Locations:** Keep sensitive ZUL files out of public reach. Place admin or internal pages under `WEB-INF` (so they can only be included programmatically, not via direct URL)【12†L79-L84】, or use ZK's authentication to guard them. Even if an attacker knows the path, they shouldn't be able to load it without going through your application's security checks.

- **Use Roles/Permissions in UI flow:** Instead of letting the client decide which page to load, make such decisions on the server with knowledge of the user's role. For example, if `view=adminDashboard` comes in, check on the server if the current user is an admin. If not, refuse or redirect. This ties into session/login security (broken access control prevention).

- **Avoid User-Supplied Templates:** Do not allow users to provide raw ZUL or template strings that get rendered. If you have a use-case like user-customizable dashboards or rich text editing, sanitize that input thoroughly or use a safe format (for rich text, use a markup like Markdown that you can safely convert to HTML server-side). Treat any such functionality as you would an HTML editor – with strict sanitization on save and on display.

- **Limit EL Capabilities:** By default, ZK's EL is restricted to property access and method calls on allowed objects (view model, composer, etc.). Avoid customizations that would broaden that. For instance, do not globally expose something like `System.getProperties()` to EL context. If you use ZK Functions or static method bindings, ensure they don't expose internals. It's generally safe out of the box, but be mindful when extending ZK.

- **Stay Updated:** Use the latest ZK version to benefit from security improvements. ZK 10 and above include enhancements like stricter URI handling (preventing some inclusion attacks)【10†L93-L96】 and the Inaccessible Widget Blocker (discussed later) for access control. Also, patch known vulnerabilities: e.g., a recent vulnerability (CVE-2022-36537) in the AuUploader servlet allowed unauthorized file access via a crafted request【5†L123-L131】 – upgrading ZK patched this. Keeping up-to-date protects you from such edge-case exploits in dynamic content handling.

By reducing overly dynamic behavior and tightly controlling what content and pages can be rendered, you eliminate an entire class of injection and inclusion problems. Essentially, **never let the user decide what code or page executes** – always mediate and validate their requests on the server.

Backend Java Security Practices for ZK Applications

While the focus is on ZUL files, the backend Java code in a ZK app plays a critical role in security. Below are practices (beyond the basics) that experienced developers should follow:

- **Server-side Input Validation & Injection Prevention:** ZK is mostly a UI framework; it won't automatically secure your database or backend integrations【12†L66-L74】. Apply standard injection prevention in your Java code:
    - Use **parameterized queries or ORM safety** for database access (to prevent SQL injection). Never construct SQL by concatenating user input【12†L69-L72】.
    - For LDAP queries, OS commands, or other interpreters, ensure inputs are sanitized or use safe APIs (e.g., prepared statements for LDAP, `ProcessBuilder` for commands with arguments).
    - Validate input lengths, ranges, and patterns. ZK's component constraints can handle some validation (e.g., `<textbox constraint="/^[A-Z0-9]{6}$/" />` for a code format), but always re-validate on the server side, as clients can bypass UI constraints.
    - Be cautious of **insecure deserialization**. If you use Java serialization (e.g., storing objects in session or caching user data to disk), use allowed-classes lists or avoid serialization of user-influenced objects. Prefer JSON/XML for data interchange and validate that as well.
- **Sensitive Data Handling:** Only send or store what is necessary:
    - **Least Exposure to Client:** Avoid sending sensitive data to the client unless absolutely needed. For example, don't embed a user's full SSN or password in the UI (even if hidden). If you have a form that needs such data, consider providing it only when the user requests or needs to view it, rather than on initial

page load.

    - **Storage in Session:** Store minimal information in the user's session. A user ID and roles/permissions are usually enough. Don't keep things like plaintext passwords, credit card numbers, or large objects in the session. Not only is it a security risk if session data leaks, but it also affects performance.

    - **Protect Files and Resources:** If your ZK app works with files (exports, reports, images), keep them in secure locations. For example, if users upload files, save them in a directory not directly accessible via URL, and stream them via a controlled servlet when needed. Similarly, if your app reads configuration or user data files, ensure they are in non-web-accessible paths and with proper ACLs on the filesystem.

    - **Masking and Encryption:** If certain data must be stored or transmitted, use encryption or masking. For instance, encrypt sensitive fields in the database. Within the UI, mask secrets (e.g., show API keys as `****1234` with an option to reveal on demand). This way, even if the UI is compromised, not everything is exposed.

- **Access Control (Enforce on Server):** Don't rely solely on the client-side to hide or disable protected features:

    - Implement robust authentication and authorization. ZK itself doesn't provide a login module【12†L75-L79】, so integrate something like **Spring Security** or container-managed security. These frameworks handle password hashing, login attempts, and can restrict URLs or method calls by roles.

    - On every server-side event that performs a sensitive action, check the user's permissions. For example, if there's an `onClick` event for an "Delete User" button bound to a method `deleteUser(userId)`, that method should verify the current user is an admin or owns that `userId` before proceeding.

    - Use **InaccessibleWidgetBlockService** (IWBS) for defense in depth. Since ZK 10.0.0, IWBS is on by default, blocking events from UI components that are not supposed to be interactive (like disabled or not rendered to the client)【10†L69-L77】. This helps prevent a malicious user from enabling a hidden button via browser dev tools and clicking it. If you're on an older ZK version, you can enable this by configuring a listener in `WEB-INF/zk.xml` or upgrading to 10+. IWBS complements your own checks by automatically rejecting illegal UI interactions at the framework level.

    - **Broken Access Control Testing:** Ensure that pages or functionalities meant for certain roles cannot be accessed by others. This might mean trying to load admin pages as a normal user (should be prevented via redirect or error) and ensuring UI components meant for admins truly don't do anything if somehow triggered by a non-admin.

- **Logging and Monitoring:** Implement logging for security-relevant events in your ZK application:

    - Log authentication events (login success/failure, logout).

    - Log access control failures (e.g., if a user tries to perform an action and is denied, record it).

    - Capture unusual input in logs (e.g., catch exceptions that might indicate attempted injection or misuse, and log the input that caused it).

    - Use these logs. Monitor them or feed them into a SIEM (Security Information and Event Management) system. This is important for detecting attacks in progress or identifying weaknesses. For example, multiple XSS attempts in logs could indicate a probing attacker, prompting a code review in that area.

- **Security Configuration:** Leverage server and framework settings:

    - Ensure you have a **secure deployment configuration** – e.g., disable directory listings on the web server, use HTTPS, set secure cookies, and if using a cluster, secure the communication between nodes.

    - If your application is large, consider a Web Application Firewall (WAF) which can provide generic protections (though be cautious as AJAX applications like ZK might not work with all WAF rules without tuning).

    - Set appropriate **HTTP headers** for responses from your ZK app: `X-Frame-Options: SAMEORIGIN` to prevent clickjacking, `X-Content-Type-Options: nosniff`, `Referrer-Policy`, etc. These can usually be added via a servlet filter or proxy server.

In summary, treat your ZK application as you would any enterprise web app: follow OWASP Top 10 practices (ZK's documentation even maps these risks to ZK specifics【10†L43-L52】) and never assume the framework alone has you covered. ZK provides many hooks and default protections, but how you use it determines the security outcome.

Session and Login Security in ZKoss Applications

Session management and authentication are fundamental to ZK application security. Here's how to address them in the context of ZK:

Session Security in ZK

ZK applications are stateful, relying on the server session to store UI state and user data. Each client browser window is associated with a ZK **Desktop** (an instance representing the page view on the server) identified by a unique desktop ID token. Maintaining session security involves both standard web practices and understanding ZK's mechanism:

- **CSRF Protection via Desktop ID:** By design, ZK is an Ajax-driven framework with no reliance on static form submissions, which inherently reduces CSRF risk【16†L83-L92】. Every ZK Ajax request includes a unique desktop ID (a GUID) that was generated when the page was loaded【16†L89-L98】【16†L109-L117】. The server knows the valid desktop ID for that session and page, and will reject requests with missing or incorrect IDs as invalid. This acts like a CSRF token – an attacker would have to guess a valid, ephemeral ID for the target's session, which is impractical. Additionally, component IDs and event references must match the server's expectations, or the request is ignored【16†L89-L98】. **Bottom line:** ZK's architecture gives a layer of CSRF defense automatically.
    - *Tip:* You can enhance this by also checking the `Origin` or `Referer` header in a servlet filter for sensitive actions (defense-in-depth to ensure the request comes from your domain)【16†L72-L80】. But this is often not necessary unless you have non-ZK endpoints.
    - For traditional multi-page flows or file download links, do implement CSRF tokens in those forms since they might not use the ZK AJAX engine.
- **Session Hijacking Prevention:**

Session Fixation: Always invalidate the old session upon successful login to get a new session ID. If using Spring Security, this happens automatically (it creates a new session and transfers attributes)【25†L1-L4】. For custom logins, explicitly do:

java
CopyEdit
```
Sessions.getCurrent().invalidate(); // invalidate ZK session
(underlying HttpSession)
Sessions.getCurrent(true);          // create a new session
```

- (If using `Executions.sendRedirect()` to post-login page, you might let the container create a new session on redirect.) The idea is to ensure an attacker cannot plant a session ID (via an insecure pre-login link) and have the user adopt it.

Secure Cookies: Configure your web container or via code to mark the JSESSIONID cookie as HttpOnly and Secure. HttpOnly prevents JavaScript from accessing the cookie (mitigating XSS stealing the session), and Secure ensures it's only sent over HTTPS. This is typically done in `web.xml` with:

xml
CopyEdit
```
<session-config>
  <cookie-config>
    <http-only>true</http-only>
    <secure>true</secure>
  </cookie-config>
  <session-timeout>30</session-timeout> <!-- timeout in minutes -->
</session-config>
```

-
- **Session Timeout:** Set a reasonable session timeout (e.g., 15-30 minutes of inactivity, depending on your app). ZK will automatically clean up desktops on session expiration. A shorter timeout limits the window for hijacking if someone walks away from a logged-in session.

- **Concurrent Sessions & Logout:** Decide if users can have multiple sessions (e.g., login from multiple devices/browsers). If not, enforce single session per user (Spring Security can control this). Also, ensure that on logout, you invalidate the session (`Sessions.getCurrent().invalidate()`) and perhaps also remove any client-side tokens or data. Encourage users to log out especially on shared computers.

- **Monitor Session for Anomalies:** In high-security contexts, you might implement session-IP binding (if IP changes mid-session drastically, flag it) or device fingerprinting. ZK doesn't do this out-of-the-box, but you can get client info via `Executions.getCurrent().getRemoteAddr()` and user-agent. Be cautious: legitimate reasons (mobile network, corporate proxies) can cause IP to change.

- **Resource Cleanup:** ZK sessions (desktops) hold a lot of state. Ensure you do proper cleanup to avoid leaking data:
    - If you store custom objects in session or desktop attributes, remove them on logout or when no longer needed.
    - Use ZK's `SessionCleanup` or `ExecutionCleanup` listeners if needed to clean sensitive info after requests.
    - If using server-push or background threads, be careful not to keep references to session-bound data longer than necessary.

Login Security in ZK Applications

Authentication is often custom in ZK, but you can integrate established frameworks:

- **Use Established Security Frameworks:** Leverage **Spring Security** with ZK for a robust solution【23†L8-L12】. There are ZK tutorials and examples for this integration. Spring Security will handle password encoding, login, logout, remember-me, session fixation, CSRF (for non-Ajax), and even UI-level method security (like `@PreAuthorize` annotations in your view model methods).
    - If using Spring Security, note that ZK events will still go through, but you can secure them by checking roles in methods or using Spring Security's annotation on methods invoked by ZK (e.g., a service called in a ZUL event).
    - Spring Security can also secure the ZUL page URLs. For example, you can configure `/admin/**` URL pattern to require ROLE_ADMIN, which means if an attacker tries to load an admin ZUL, they get redirected to login automatically.
- **Custom Login Implementation:** If not using a framework, do it carefully:
    - **Password Handling:** Always store passwords hashed (e.g., BCrypt). On login, hash the input password and compare with stored hash. Never store or log plaintext passwords. Use a strong hashing algorithm – this is critical if your database leaks.
    - **Login Form Security:** Use `<textbox type="password"/>` for password fields so it's masked. ZK will treat it as a normal input; ensure the form is submitted over TLS. You might have a login button that triggers an event to verify credentials. On failure, give a generic error ("invalid login") to avoid username enumeration.

    - **Brute Force Mitigation:** Implement a counter for login attempts. For example, after 5 failed tries for a user account, lock it for 15 minutes. Or use a CAPTCHA on the login form after a few failures. This can be implemented in your auth service logic.

    - **Logging:** As mentioned, log login attempts (especially failures with the username, timestamp, and source IP). This helps detect brute force or credential stuffing attacks.

    - **Post-Login:** After login, set a session attribute like `session.setAttribute("user", userObject)` to mark the user as authenticated. Use this in access checks on pages (as shown in ZK Essentials: checking a `UserCredential` in session)【26†L61-L69】【26†L75-L83】. Then redirect to the main page. (Remember to invalidate the previous session if you haven't yet.)

    - **Logout:** Provide a logout function that calls session invalidate. In ZK, you might have a "Logout" button that does `Sessions.getCurrent().invalidate(); Executions.sendRedirect("/login.zul");`. Once invalidated, the ZK desktop is destroyed and any further action requires a new login. Also clear any client state if you stored something in cookies (for example, if you implemented a "remember me" cookie yourself, remove it).

    - **Session Transfer on HTTPS Downgrade:** Ensure the app doesn't allow switching to HTTP after login. If a user logged in over HTTPS, they should not continue the session over HTTP (or the cookie could be intercepted). Ideally force all traffic over HTTPS.

- **Protecting Pages and Components:** Even with login done, ensure that each page verifies the user's role:

Use a filter or a check at page load (composer or view model initialization) to redirect unauthorized users. For instance, in an admin page's controller, do:

java
CopyEdit
```
if (!user.isAdmin()) {
   Executions.sendRedirect("/forbidden.zul");
   return; // stop further setup
}
```

    - This prevents an authenticated but non-admin user from seeing admin content if they somehow reach the page.

    - On the client side, you might hide menu options for unauthorized users, but remember those are just cosmetic. The real enforcement is on the server when the page or action is attempted.

- **Two-Factor Authentication (2FA):** For advanced security, consider adding 2FA for login. This could be integrated via Spring Security 2FA modules or custom code (sending an OTP via SMS/Email and prompting the user). If implemented, treat the OTP input like any sensitive input (validate it, rate-limit attempts, etc.). ZK can easily create a UI for OTP entry.

- **Error Handling:** Be mindful of error messages during login. If using ZK's `Messagebox` to show errors, that's fine – just don't reveal, for example, "User exists but password is wrong" vs "User not found". Keep it generic to not give away whether a username is valid.

Security Tips for ZK Components and Data Exposure

Finally, here are additional tips for safe usage of specific ZK components and patterns that often arise in advanced ZK development:

- **Use `<zscript>` Sparingly:** We covered script injection risks – in general, keep business logic out of ZUL `<zscript>` blocks. Use Java composers or view models to handle logic and call secure backend services. This not only improves security (less inline code to audit) but also maintainability.

- **Component Visibility vs. Existence:** To secure UIs, prefer not to render components at all rather than hiding them. For example, use `unless` attribute to conditionally create a component only for authorized users, instead of creating it and setting `visible="false"`【27†L39-L47】【27†L43-L50】. If it's never rendered to the client, an attacker can't force it visible. If you must create and hide (maybe for dynamic UI reasons), then rely on IWBS and server checks to prevent misuse.

- **Components that Hold Data:** Some components can store data that might not be obviously visible:

    - **Listbox/ListModel:** If you use a `ListModelList` or other models, by default ZK only renders the needed items to the client. However, if the list is small or not using paging, a user could view source or use ZK's client-side widget API to see list items' labels. Ensure that any sensitive info in list items (like an email or ID) is something the user is allowed to see. If not, don’t include it in the model or mask it.

- **Component Attributes:** ZK components allow you to set custom attributes (e.g., `comp.setAttribute("secret", value)`). These are server-side only by default and not sent to the client. However, be careful: if you set an attribute and then later use it in a client-side context (like `Clients.showNotification(comp.getAttribute("secret"))`), you might accidentally expose it. Treat server-side stored secrets with care and remove them when done.

- **Serialization to Client:** Some data may serialize to the client as part of component state. For instance, if you use certain ZK features like client binding or add custom properties to components via `setAttribute` with `Component.COMPONENT_SCOPE`, they might be marshaled to the client. Avoid attaching sensitive data directly to UI components in a way that could go to the browser. When in doubt, test what is visible on the client (ZK’s developer tools or browser’s dev tools can show you the HTML/JSON being sent).

- **File Upload/Download Components:** Use ZK’s `<upload>` with care. Always validate the uploaded file on the server side (check file type, size, scan for viruses). For file downloads, ensure the user is authorized to get the file. ZK’s `Filedownload.save()` makes it easy to send files, but you should verify access before calling it. Keep the repository of files secure (ideally outside webroot or with random file names so they can’t be directly URL-accessed).

- **ZK Charts/Graphs and Data:** If using ZK charts or other components that render data on client side (like Google Maps component, etc.), be mindful that any data you feed them will be on the client. E.g., if a chart has data series values that are sensitive, those values end up in the page source or AJAX responses. Aggregate or anonymize data if needed when sending to client.

- **Desktop/Application Memory:** ZK keeps state on the server, which means memory usage grows with each session. From a security perspective, be aware of **DoS via memory exhaustion**. An attacker opening many sessions or causing large components to be created could try to overwhelm the server. Mitigations:

  - Set `max-desktops-per-session` and `max-requests-per-session` in `zk.xml` to reasonable limits to prevent abuse【29†L141-L149】.

  - Use `AuUploader` and file size limits to prevent huge file uploads from consuming memory.

    - Consider using ZK's server push carefully; limit the number of concurrent pushes or use queuing to avoid floods.
    - Monitor your application's memory and have alerts if it's growing abnormally (could indicate a denial of service attempt or just a memory leak bug).
- **Testing and Updates:** Regularly test your ZK application with security tools. Use OWASP ZAP or Burp Suite to proxy your app and scan for issues like XSS, missing headers, etc. Although automated scanners might not navigate ZK's AJAX easily, you can script some interactions or test specific requests. Also, keep an eye on ZK's release notes for security advisories. For example, if a new ZK version fixes a vulnerability and you're on an older version, assess the risk and update promptly【12†L49-L52】【10†L47-L53】.

By following the above practices and being mindful of how ZK components handle data and execution, you can build a robust, secure ZKoss application. Security is an ongoing process: incorporate these checks into your development and deployment pipeline, and stay educated on new threats and patches. With careful coding and configuration, ZK applications can be made as secure as any traditional web application while providing a rich user experience.

Security Issues in ZKoss (ZUL) Applications: Identification and Mitigation

Introduction

ZKoss (ZK) is a rich Java-based web framework that uses **ZUL (ZK User Language)** files for UI definition. While ZK simplifies building UIs, developers must be vigilant about security. This report covers common vulnerabilities specific to ZUL files and advanced backend practices to secure ZK applications. We focus on identifying issues like script injection, cross-site scripting (XSS), and unsafe use of expressions, along with session/login security. Real-world ZUL examples, detection steps, and mitigation strategies are provided for each case.

Common Vulnerabilities in ZUL Files

1. Script Injection Vulnerabilities (`<zscript>` Blocks)

ZUL files allow embedding server-side scripts using the `<zscript>` element. By default, ZK uses Java for `<zscript>`, but it can run other languages (Groovy, JavaScript, Ruby, Python) via the `language` attribute

zkoss.org
. **Script injection** occurs when an application executes dynamically constructed code that includes untrusted input, leading to **remote code execution (RCE)**. Groovy scripts in `<zscript>` are particularly sensitive because Groovy is dynamic and powerful.
Example Vulnerable ZUL (Groovy Injection):

xml
CopyEdit
```
<window title="Admin Console">
  <zscript language="Groovy">
    // BAD: Evaluating user-provided script (highly insecure)
    def userScript = Executions.getCurrent().getParameter("code");
    if (userScript != null) {
        new GroovyShell().evaluate(userScript);  // Vulnerable:
executes arbitrary Groovy code
    }
  </zscript>
</window>
```

- In this example, an attacker who passes a malicious `code` parameter (e.g., `System.exit(0)` or spawning a shell) can execute arbitrary server-side code. Using `GroovyShell.evaluate()` on a dynamic string is a known code injection sink github.com

  docs.datadoghq.com
  .

- **Identification (Detection Steps):**
  - **Search ZUL files for `<zscript>` usage** – Review all occurrences of `<zscript>` and note the `language`. Pay extra attention to `language="Groovy"` or other scripting languages. These indicate places where server-side code is embedded.
  - **Examine for dynamic code execution** – Within each `<zscript>` block, look for use of classes like `GroovyShell`, `GroovyClassLoader`, `Interpreter`, or eval/execute methods. Also flag any use of `Executions.create**` methods that compile or run dynamic content. For Java `<zscript>`, check for use of reflection or `Executions.evaluate()` (if any).

  - **Trace input sources** – If the script constructs code from variables or parameters, determine if any of those come from user input (e.g., `Executions.getCurrent().getParameter`, component values, or session attributes set by users). Any concatenation of user data into code is a red flag.

  - **Real-world clues** – Often, script injection issues arise in admin or customization features (e.g., allowing admin to enter scripts for custom logic). Identify any such feature in documentation or code comments and review it thoroughly.

- **Mitigation Strategies:**

  - **Avoid Dynamic Evaluation:** The safest approach is not to evaluate user-supplied code at all. Implement required logic in Java (controllers or view models) rather than via dynamic Groovy scripts. This keeps code static and compiled, eliminating this injection vector.

  - **Restrict Scripting Capabilities:** If dynamic scripts are absolutely needed (e.g. plugin systems), do not expose them to untrusted users. Require authentication and limit this feature to admin roles. Additionally, sandbox the script environment: disable dangerous classes/methods. (For example, configure a custom `ClassLoader` or use Groovy's secure features to whitelist permitted operations.)

  - **Input Validation:** Rigorously validate or sanitize any input that might be used in script. *Never* pass raw user input to `GroovyShell.evaluate()` or similar. If you must accept expressions (say, a math formula), parse and validate the format before evaluation.

  - **Use FindBugs/CodeQL rules:** Employ static analysis tools which have detectors for Groovy script injection. For instance, FindSecBugs has rules to catch usage of GroovyShell w
    github.com
    nput【4†L205-L212】. This can help flag vulnerable code during development.

  - **Least Privilege:** Run the application with minimal privileges. Even if code injection occurs, a locked-down security manager or low-privileged OS user can limit damage. (Advanced setups might remove the Groovy engine entirely from production if not used.)

By treating any `<zscript>` as potential executable code and avoiding inclusion of unvalidated data, you prevent attackers from injecting scripts that the server would run.

2. Cross-Site Scripting (XSS) in ZUL Content

Cross-site scripting is a pervasive web vulnerability where malicious scripts are injected into pages v

zkoss.org
hers【1†L56-L64】. In ZK, XSS typically arises when **user input is included in ZUL views without proper encoding**. The ZK framework encodes many outputs by default (to help developers), but there are exceptions where developers must be cautious.
How ZK Handles Encoding: ZK automatically escapes special characters (`<`, `>`, `&`, etc.) for all standard input components and label outputs. For example, using a value binding in a textbox or label is safe:

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

- Even if `any_value` contains `<script>alert('XSS')</script>`, ZK will escape it (the `<`
zkoss.org
lt;`, etc.)【1†L68-L76】. In fact, an EL expression in a ZUL page implicitly renders a L
zkoss.org
is encoded【1†L81-L84】. **This means normal data binding to UI components is generally safe from XSS by default.**

- **Unsafe Cases (When ZK Does *Not* Encode):** Certain ZK components and techniques deliberately allow raw HTML for flexibility. These will **not auto-escape** content and thus can introduce XSS if misused:

`<html>` Component (Html.setContent) – Intended to embed raw HTML.
zkoss.org
encoded【29†L127-L136】. Example:

xml
CopyEdit
```
<html>${userComment}</html>
```

  - If `userComment` includes a script tag, it will be rendered as-is, executing malicious script. (Note: Since ZK 10.0.0, the Html component attempts to sanitize content by default to reduce XSS risk, but developers are advised *not* to rely on this and never include raw `<script>
    zkoss.org
    vided content【2†L103-L107】.)

- `<comboitem content="...">` – Comboitem's content attribute is raw HTML
  zkoss.org
  ed【29†L127-L135】. Any user-supplied HTML here can execute scripts or HTML.

- **Native HTML in ZUL (`<div xmlns="xhtml">` or `xmlns:native`)** –
  Embedding native HTML tags directly in ZU
  zkoss.org
  ZK's encoding【1†L87-L95】. E.g., `<div xmlns="native">${data}</div>`
  will inject raw data.

Client-side Utility Methods – Using `Clients.showNotification(String)`,
`Clients.alert(String)`, or similar to display messages, and
`Clients.evalJavaScript(String)` to run scri
zkoss.org
not encode their input by design【2†L113-L120】. They allow formatting or direct script execution on the client. If you pass user data directly, it can inject scripts. For example:

java
CopyEdit
```
// In a ZUL event listener or ViewModel:
String name = userService.getName();           // could be
attacker-controlled
Clients.evalJavaScript("showWelcome('" + name + "')");
```

- If `name` contains a quote or `</script>`, it could break out of the string and run arbitrary JS. Similarly, `Clients.showNotification("Success: " + userInpu&#8203;:contentReference[oaicite:12]{index=12}er any HTML in` userInput`【2†L113-L120】. Developers must manually escape or sanitize in these cases.

- **Page Directive Attributes** – All attributes in `<?page?>` (the ZUL pa
  zkoss.org
  are output without encoding【2†L129-L137】. In practice, if any page attribute is set from user data (rare, usually static design-time values like title), it could be an injection vector.

Example Vulnerable ZUL (XSS):

xml

CopyEdit
```
<window title="Feedback">
  <!-- Developer wants to allow some HTML in userComment -->
  <html>${param.userComment}</html>
  <button label="Preview"
onClick='Clients.showNotification(param.userComment, "info")'/>
</window>
```

- Here, if `param.userComment` is `<img src=x onerror="alert('XSS')">`, the `<html>` component will inject an actual image tag with the onerror script, executing the alert. The `Clients.showNotification` will also display it un
  zkoss.org
  ich could pop the alert or distort the UI【2†L113-L120】.

- **Identification (Detection Steps):**

  - **Find raw HTML uses** – Grep for `<html>` components or `.setContent(` calls in Java. These are likely points where raw output is used. Check if the content comes from user input (e.g., form fields, database content that users can influence).

  - **Check comboitem and others** – Search for `comboitem content=` in ZULs. Also look for usage of native namespaces (`xmlns:native` or `xmlns:xhtml`) which signal direct HTML embedding.

  - *Scan for Clients. calls** – In ZUL onClick/ onOK handlers or controllers, find calls to `Clients.evalJavaScript`, `Clients.showNotification`, `Clients.alert`, etc. **Audit what data is being passed**. If any parameter in those calls originates from a user (request param, component value, etc.) and isn't encoded, that's a potential XSS.

  - **Identify disabled encoding** – Although ZK encodes labels by default, ensure no code explicitly disables encoding. For instance, in older ZK, `Label.setRawValue()` or `disableClientEscaping` flags might output raw HTML. Such patterns should be treated as sinks for XSS.

  - **Pen-test dynamic content** – Run the app and input typical XSS payloads (e.g., `<script>alert(1)</script>` or `<img onerror=alert(1) src=nonexistent>` in form fields) and see if they are rendered in any response/UI without being neutralized. This dynamic testing complements code review to catch any missed spots.

- **Mitigation Strategies:**
  - **Prefer Safe Components**: Use ZK's normal components (Label, T
    zkoss.org
    for user-generated content, as they encode dangerous characters by default【29†L117-L125】. For example, display user comments in a `<label>` or `<textbox readonly="true">` rather than an `<html>` element. If you simply need multiline text display, ZK's Label supports multiline (or use `<label multiline="true">` which still escapes HTML).
  - **Sanitize if Raw HTML Needed**: If your use-case requires allowing some HTML (e.g., formatting in user comments), sanitize it on the server before rendering. Use libraries like **OWASP**
    zkoss.org
    r, **Jsoup**, or ZK's own utility `XMLs.escapeXML(String)`【2†L133-L140】 to strip or encode disallowed tags. *Example:* before setting content on an `<html>` component, do:
    `htmlComp.setContent(XMLs.escapeXML(userComment));` – this will convert `<` to `&lt;`, neutralizing any tags.
  - **No Scripts in HTML Content**: Neve
    zkoss.org
    ipt>`tags via user input. Even if ZK 10+ auto-sanitizes`<html>`content【2†L103-L107】, it's wise to explicitly remove scripts or event attributes to be sure. **Tip:** If you must allow limited HTML (e.g.,`<b>`, `<i>\`), whitelist those and strip everything else.

Escape Data in Client Calls: For `Clients.evalJavaScript`, don't inline user strings into JavaScript code. If you need to pass user data to a JS function, use JSON encoding or base64. For example:

java
CopyEdit
```
String safeJson = org.zkoss.json.JSONValue.escape(userInput);
Clients.evalJavaScript("showMessage(JSON.parse('" + safeJson + "'))");
```

- Better, use ZK's `JavaScriptValue` (if available) or send the data to client as an argument rather than constructing JS. For notifications, if you want to display raw text,
  zkoss.org

howNotification(Strings.escape(userInput))`or supply the`raw=true` parameter appropriately after sanitizing.

  - **Review ZK Encoding Gaps**: Be aware of which propertie
    zkoss.org
    ded【29†L126-L134】. For instance, avoid binding user data to `Html.setContent` or `Comboitem.setContent` unless sanitized. ZK's documentation explicitly warns these are not encoded and need developer care【29†L129-L137】. Incorporate these guidelines in code reviews.

  - **Content Security Policy (CSP)** (Advanced): Consider using a strict CSP in your HTTP response headers to mitigate impact of any XSS that slips through. For example, disallow inline scripts except those from your domain. This is a second line of defense.

By encoding all user-supplied content or sanitizing in cases where raw HTML is needed, you can effectively prevent XSS in ZK applications. Remember that ZK's default is safe for most UI elements – problems usually arise when deliberately bypassing those safeties for flexibility.

3. Improper Use of EL Expressions and Dynamic Content Rendering

ZK supports powerful expression language (EL) binding and dynamic UI composition. **Improper use of these features can introduce security issues** such as unauthorized page inclusion, template injection, or expression evaluation risks. This category covers scenarios outside of simple XSS – for example, including the wrong ZUL page based on user input, or using EL in a way that leaks data or calls unsafe methods.

Unsafe Dynamic Page Inclusion (ZUL Path Injection): Developers sometimes compose pages dynamically. For instance, using a parameter to decide which ZUL to include:

xml
CopyEdit
```
<!-- Vulnerable if 'page' param is not validated -->
<include src="/views/${param.page}.zul"/>
```

- If `param.page` is attacker-controlled, this can load arbitrary Zul files. An attacker might access pages they shouldn't (e.g., `admin` or `../WEB-INF/config`). While the servlet spec prevents loading files from WEB-INF via direct URL, an include might still pull it if not protected (depending on ZK implementation). Even if not, an attacker could try values that cause unintended behavior or errors (possibly revealing stack traces or sensitive info). In general, **treat any user input that constructs a file path as dangerous** (similar to directory traversal vulnerabilities).

Template Injection via `Executions.createComponents`: ZK's API allows creation of components from ZUL content at runtime:

java
CopyEdit
```java
String zulPath = Executions.getCurrent().getParameter("zulPage");
Executions.createComponents(zulPath, parentComponent, null);
```

- If `zulPage` is user-supplied (query param or form), an attacker could input the path to an admin page or a malicious Zul file on the server. Even worse, if the app ever takes raw ZUL markup from a user and calls `createComponentsDirectly(String zulMarkup, ...)`, that's effectively evaluating user code (similar to an HTML template injection leading to XSS or RCE depending on content). This is a serious issue – it's like allowing an attacker to feed your server a new ZUL page to render.

Improper EL Usage: ZK's EL in bindings is mostly safe because it doesn't eval arbitrary expressions from the client – the developer defines the expression in the Zul. However, *if* developers abuse it, e.g., by constructing EL strings dynamically or using it to call static methods unsafely, it can bite. For example, if a binding calls a static method that should be restricted:

xml
CopyEdit
```xml
<label value="${systemProperties['os.name']}"/>
```

- This might expose system info. Or using ZK's older scripting, one might do something like `${Class.forName(param.classname)}` which could trigger classloading. Such uses are uncommon but illustrate that EL can access server-side resources if not careful. The key point is **never feed untrusted input into EL evaluation**. ZK doesn't expose an eval EL function by default, so this is more about design: ensure your ZUL bindings don't inadvertently expose sensitive data or operations.

Example Issue (Dynamic Include):

xml
CopyEdit
```xml
<window apply="com.app.MyComposer">
  <!-- Show different page based on 'view' request parameter -->
  <zscript>
     String page = Executions.getCurrent().getParameter("view");
     if(page != null) {
         Include include = new Include("/zul/" + page + ".zul");
         self.appendChild(include);
     }
  </zscript>
</window>
```

- If an attacker accesses `?view=../../WEB-INF/sensitive`, this code will try to load `/zul/../../WEB-INF/sensitive.zul`. This might throw an error or potentially load a forbidden file if traversal is not blocked. Even if not, it allows scanning for valid ZUL pages or loading pages out of intended flow (e.g., loading an admin.zul without going through login). This is an **access control issue** as well as a possible injection issue (if a crafted path breaks out of the expected directory).

- **Identification (Detection Steps):**

  - **Audit includes and navigation logic** – Search for `<include src="${...}">` in ZUL or any usage of `Include` components in code. Also search Java code for `Executions.createComponents`, `Executions.createComponentsDirectly`, `Executions.include` or similar. Wherever a path is constructed dynamically, flag it.

  - **Trace input for dynamic paths** – If a dynamic include uses a variable, find where that variable comes from. Is it a request parameter, session attribute, or a hardcoded mapping? If it's user-controlled and not validated against an allow-list of safe values, it's a potential vulnerability.

  - **Look for eval-like patterns** – Although ZK's binder doesn't provide an eval from string, check if the code uses any evaluation utilities (perhaps using `javax.el.ELProcessor` or OGNL). Also, check `<zscript>` blocks for usage of `Executions.evaluate()`, `Path.getComponent()` with dynamic expressions, or custom EL resolvers that might be abused.

  - **Examine data-binding expressions** – Review your MVVM bindings (in `viewModel` and data bindings) for any suspicious expressions that might expose more than intended. E.g., using `${vm.someMap[param]}` where `param` could be user input – ensure it can't be abused to get arbitrary map keys. Or if using **ZK Functions** or static method calls in EL, ensure they can't do something unsafe. It's rare to find critical issues here, but a quick check can reveal oversights.

  - **Test scenario** – Try manipulating any request parameters that seem to select pages or templates. If you find pages that aren't meant to be user-facing can be loaded by direct URL or parameter, that's a broken access control. Ensure unauthorized users cannot reach admin or internal ZULs by guessing paths.

- **Mitigation Strategies:**

Validate and Whitelist: Never use raw user input to select a ZUL page or template. If you need to switch views based on input, maintain a **whitelist map** of allowed page names. For example:

java
CopyEdit
```
Map<String,String> pageMap = Map.of(
    "home","/zul/home.zul",
    "dashboard","/zul/dashboard.zul"
);
String page = param("view");
if(pageMap.containsKey(page)) {
   Include include = new Include(pageMap.get(page));
   // ...
} else {
   // invalid page requested
}
```

  - This ensures only known good pages load. Any attempt to brea
    zkoss.org
    `../`) will simply not match the whitelist.

  - **Keep ZUL files out of public reach**: Place sensitive ZUL files (like admin pages) under `WEB-INF` or configure your web.xml to forbid direct access. While ZK pages are typically served through the ZK servlet and not directly as files, it's wise to treat them as server-side view templates. If a page should only load after login, do not allow it to be included without proper checks.

  - **Use Executions.createComponents safely**: If you use `createComponents` with a path, follow the whitelist approach above. If using `createComponentsDirectly` with a template string, avoid using user-provided template data entirely. That method is meant for cases like generating a small snippet of components – feed it only constant or server-generated strings. *Do not pass user text into it.* If you must allow users to define templates (e.g., custom form designer), heavily sanitize that input and strip any `<zscript>` or dangerous elements from it before evaluation.

- **Avoid Overly Dynamic EL**: In data binding expressions, stick to property accesses and method calls you control. Avoid exposing general maps or scripting objects that could be misused. If you use something like ZK's FunctionMapper to call static methods, restrict it to safe utilities. And absolutely do not design a system where end-users input EL expressions – ZK has no such feature by default, so don't add one without sandboxing.

- **Framework Patches**: Stay updated on ZK security patches. For instance, ZK 10 introduced stricter handling of GET/POST to prevent misuse of the servlet【10†L93-L96
  zkoss.org
  ed `InaccessibleWidgetBlockService` by default to block client-altered component access【10†L69-L77】. Use the latest ZK version so that known vulnerabilities in dynamic content handling (e.g., old file upload issues, path traversal bugs)
  nvd.nist.gov
  or example, a recent CVE-2022-36537 was patched to prevent an attacker from accessing files via the AuUploader component by crafting requests【5†L123-L131】【5†L123-L125】. Keeping up-to-date shields you from such framework-level flaws.

By reducing overly dynamic behavior and tightly controlling what content and pages can be rendered, you eliminate an entire class of injection and inclusion problems. Essentially, **never let the user decide what code or page executes** – always mediate and validate their requests.

Backend Java Security Practices for ZK Applications (Intermediate to Advanced)

While the focus is on ZUL files, the backend Java code in a ZK app plays a critical role in security. Here we outline some practices (beyond the basics) that experienced developers should follow:

- **Server-side In
  zkoss.org
  on & Injection Prevention:** ZK is mostly a UI framework; it won't automatically secure your database or backend integrations【12†L66-L74】. Apply standard injection prevention in your Java code:

  - Us
    zkoss.org
    rized queries or ORM protections** for database access (prevents SQL injection).

Never concatenate user input into SQL directly【12†L69-L72】.

- For LDAP queries, OS commands, or other interpreters, similarly ensure inputs are sanitized or use safe APIs.
- Validate inputs rigorously using whitelist patterns. ZK provides a `Constraint` mechanism on input components for basic validation, but also enforce checks on the server side (e.g., using Jakarta Bean Validation or manual checks) to ensure malicious data is rejected.
- Beware of **insecure deserialization** if you use Java serialization (e.g., storing user-provided objects in session or cache). This is more a concern if you accept serialized data or use libraries that might do so. Use modern serialization alternatives (JSON, etc.) and never deserialize unknown data.

- **Sensitive Data Handling:** Only send to the UI what is necessary for the user. Because ZK keeps components and data in server memory, you might be tempted to attach lots of data to components. Remember:
  - **Do not store secrets in component attributes** that end up sent to client. For example, avoid `<label value="${user.password}"/>` – you wouldn't do that plainly, but even storing it in a non-visible attribute could be risky if the client can somehow see it. Keep sensitive info server-side (in session or security context).
  - If you have to display partial sensitive data (like email or ID), mask it (e.g., show only part of it).
  - Never expose internal file paths, config values, or stack traces in the UI. Catch exceptions and show generic errors.
  - Store files (like exports or reports) in non-web accessible locations (e.g., under WEB-INF or outside web root)【12†L79-L84】, and stream them to the user after permission checks rather than linking directly.
- **Access Control (Enforce on Server):** ZK's UI controls (like disabling a button or hiding a component) are not sufficient for security. Always check permissions on the server before performing an action:
  - Use a robust authentication and authorization system. ZK itself doesn't provide login/auth – integrate with **Spring Security** or container security【12†L75-L79】. These frameworks can handle session management, password hashing, and URL/page access rules. ZK pages can be protected via filters or configured to

require certain roles.

  - On every sensitive server operation (event lis
    zkoss.org
    mand), verify the user's role/rights. For example, if a button deletes a record, the event handler on the server should ensure the current user is allowed to delete that record, even if the UI hid the button for others. This protects against forged requests or clients manipulating the UI.

  - Use ZK's `InaccessibleWidgetBlockService` (enabled by default since 10.0.0) which **blocks events from components that are not supposed to be usable (like disabled or invisible ones)**【10†L69-L77】. This is a nice safety net: if an attacker enables a hidden admin button via browser dev tools, ZK will ignore the click on the server. Earlier versions of ZK didn't have this on by default; if you use an older version, consider e
    zkoss.org
    anually via a listener in `web.xml` as documented by ZK. This service complements your own access checks.

  - **Broken Access Control check**: Ensure that pages that require login cannot be loaded without login. For instance, if using a custom login, each ZUL page's controller (composer or viewModel) can redirect unauthenticated users to the login page in its initialization if needed. Or use a servlet filter to intercept requests to `/secure/*` ZULs. This prevents direct access by guessing URLs【26†L61-L69】.

- **Session Management:** Proper session security keeps user sessions from being hijacked or misused:

Session Fixation: When a user logs in, invalidate any old session and issue a new one. If you use Spring Security, this is handled for you (it by default creates a new session on authentication)【25†L1-L4】. If not, manually do:

java
CopyEdit
```
HttpSession oldSession = request.getSession(false);
if(oldSession != null) { oldSession.invalidate(); }
HttpSession newSession = request.getSession(true);
// copy necessary attrs to newSession if needed
```

  - This ensures an attacker cannot use an existing session ID (from before login) to hijack the session.

  - **Secure Cookies:** Configure your session cookie with `HttpOnly` and `Secure` flags (usually in web.xml or container settings). HttpOnly prevents client-side scripts from reading the cookie, and Secure ensures it's only sent over HTTPS. This is critical to mitigating XSS (stops it from stealing session cookies) and safeguarding against network sniffing.

  - **Session Timeout:** Set a reasonable session timeout in web.xml (or programmatically). Inactivity timeouts reduce the window for an attacker to reuse a stolen session. For high-security apps, consider an absolute timeout (max session lifespan) or implement re-authentication for critical actions.

  - **Concurrent Session Control:** Depending on requirements, you might limit users to one session at a time or log out previous sessions on new login, etc., to reduce session abuse. Spring Security can do this; otherwise manage session IDs in a server-side store.

  - **Session Storage:** Store minimal data in the session. Keep only identifiers and necessary info (e.g., user ID, roles). Avoid placing large objects or sensitive data
    zkoss.org
     it increases risk (and memory usage). Also, if using session clustering, test that serialized session data doesn't include anything sensitive unexpectedly.

- **Login Security:** Ensure that the authentication process itself is secure:

  - **Hash Passwords:** On the server side, never store plaintext passwords. Use strong hashing (e.g., bcrypt) with salt for any password storage. On login, compare hashed values. This is outside ZK's scope but crucial if you implement your own auth.

  - **Transport Security:** Always use HTTPS for login pages and any page with sensitive data. ZK apps often are behind a TLS termination, but double-check that credentials aren't sent over plain HTTP. If using ZK's AJAX for login (e.g., submitting a form via an event), ensure the page is loaded on HTTPS and check the network calls.

  - **Brute-force protection:** Implement account lockou
    zkoss.org
    tial backoff on failed logins. ZK doesn't do this inherently, but you can track in your auth service. For
    zkoss.org

zkoss.org
ttempts, lock the account for a period.

- **Multi-factor Authentication:** For advanced security, integrate MFA for login if the application warrants it. ZK can integrate with whatever solution (since the backend is Java).

- **Logout and Session Invalidation:** Provide a logout function that truly invalidates the session (use `Sessions.getCurrent().invalidate()` or `sessi&#8203;:contentReference[oaicite:31]{index=31}e()`). Also consider the **"logout everywhere"** scenario if user requests (invalidate all sessions of that user).

- **Other Best Practices:**

  - **CSRF Protection:** ZK's architecture makes CSRF attacks more difficult by using unique component/desktop IDs in every
 zkoss.org
89-L98】【16†L109-L117】. An attacker would have to guess a moving target ID, which is impractical. However, if your app has any non-AJAX endpoints (e.g., file upload servlets, REST endpoints), protect those with CSRF tokens. ZK itself suggests using the **Synchronizer Token Pattern** with a hidden token in forms if needed【16†L109-L117】. Ensure any custom AJAX endpoints or custom servlets in your app also check origin or require a token.

  - **Clickjacking Prote
stackoverflow.com
et `X-Frame-Options: DENY` or `SAMEORIGIN` on responses (or CSP frame-ancestors) to prevent your ZK pages from being iframed by malicious sites. This isn't ZK-specific but important if your app has sensitive UI operations.

  - **Security Headers:** In addition to CSP and X-Frame-Options, consider `X-XSS-Protection` (for older IE), `X-Content-Type-Options: nosniff`, etc., via a servlet filter. These just add extra protection layers in browsers.

  - **Logging and Monitoring:** Implement robust logging for security-related events: log logins (and failures), important user actions (especially admin actions), and suspicious inputs (e.g., catches of exceptions that indicate attempted attacks). Monitor these logs. Set up alerts for multiple failed logins or unusual behavior. This helps detect if someone is probing your ZK application for vulnerabilities.

  - **Use Updated Components/Libraries:** Keep the ZK framework and any add-ons up to date. As noted, using the latest ZK version protects you from known issues (such as the file upload CVE). Also update dependent libraries (e.g., if you use

Commons FileUpload, etc.). Outdated components are one of OWASP Top risks【10†L46-L53】.

  - **Penetration Testing:** Periodically have security testing done on the application. Tools can static-scan your code for known vulnerable patterns (including the ZUL files) and dynamic scanners can try XSS, injection, etc., in running applications. Given ZK's rich interface, some automated scanners might not navigate it fully, so consider manual pen-testing focusing on the points discussed: try injecting scripts in form inputs, attempt to bypass UI restrictions, force error messages, and see if any sensitive data leaks.

Session and Login Security in ZKoss Applications

(Session management was partly covered above, but here we emphasize specifically within the context of ZK.)

ZK applications are stateful by nature – user interactions are managed via server-side sessions and the framework's **Desktop** (which is like a UI session token). As such, session and authentication security are critical.

Session Security in ZK

ZK leverages the Java EE session (`HttpSession`) to store component state and user data. Each browser tab or window corresponds to a ZK **desktop** identified by a unique ID, which ZK uses to tie AJAX requests to the correct session and page【16†L89-L98】.

- **CSRF Mitigation via Desktop ID:** Each ZK page load generates a unique **desktop ID** token that is automatically sent with every AJAX request【16†L109-L117】【16†L89-L98】. This functions similar to a CSRF token – if an attacker tries to forge an AJAX request, they would need this desktop ID, which is hard to guess. ZK also checks server-side that the component IDs in requests match those orig
  zkoss.org
  ered, preventing an attacker from invoking actions on components that weren't present or have been removed【16†L89-L98】. This means out of the box, ZK has some CSRF resistance. Nonetheless, **do not become complacent**: if you have endpoints outside ZK's AJAX (file upload, etc.), protect them as mentioned earlier. And you can reinforce security by also checking the `Origin/Referer` on incoming requests【16†L72-L80】 or adding your own CSRF token in forms if needed (especially for traditional forms that might post outside the ZK event mechanism).

- **Session Hijacking and Hardening:**

    - As discussed, always invalidate the session on login to prevent fixation. If using ZK Spring Security, this is handled for you【25†L1-L4】.
    - Make sure JSESSIONID cookie is marked HttpOnly and Secure. This is typically done in your container or `web.xml` (`<cookie-config>`).
    - ZK does not transmit the session ID in URLs by default (it uses cookies unless configured otherwise). Avoid URL rewriting (disable `URLSessionTracking` unless absolutely needed) to not expose session IDs.
    - If your app is high-value, consider implementing detection of session hijacking – e.g., if one session ID suddenly comes from a new IP or user agent, you might invalidate it. This can be complex (and can hit legitimate cases like mobile network switching IPs), so implement carefully if at all.
    - Educate users to log out and close the browser, especially on shared computers, to destroy the ZK desktop and session.
- **ZK Specific Session Tips:**
    - **Max Desktops/Requests**: ZK provides configurations to limit how many desktops (tabs) a session can create and how many requests per session to prevent DoS【29†L141-L149】【29†L135-L139】. In `zk.xml` or `WEB-INF/zk.xml` config, you can set `<max-desktops-per-session>` and `<max-requests-per-session>` to sane values to mitigate abuse (e.g., a malicious script opening hundreds of tabs or flooding requests). Consider adjusting these if your app is susceptible to such misuse.
    - **Server Push**: If you use server-push or WebSockets, e
      [hawkchen.gitbooks.io]()

      [hawkchen.gitbooks.io]()
      sent to clients are properly authenticated/authorized on the server side. Don't broadcast sensitive data to sessions that shouldn't receive it. ZK's event scope should handle this, but double-check your push event code.

Login Security in ZK

Since ZK delegates authentication to the developer, secure login flows are crucial:

- **Protecting the Login Page**: If the login is a ZUL page with a form (username/password fields), deliver that page over HTTPS. Use ZK's features (like `<textbox`

`type="password">`) to ensure the password input is masked on UI. There's no built-in encryption of the password on submit (that's what HTTPS is for), so standard web best practices apply.

- **Using Spring Security or JAAS**: Instead of reinventing login, prefer using **Spring Security integration with ZK**【23†L8-L12】. ZK has documentation on integrating it【23†L8-L12】, which allows you to use all of Spring Security's features (BCrypt password encoding, remember-me tokens, method security, etc.). This also makes things like session fixation protection, CSRF token on non-AJAX forms, and authentication flows much easier to manage.

- **Custom Login**: If implementing manually, as shown in many ZK examples, do not follow them blindly when it comes to password handling. For instance, some tutorial code compares plaintext passwords directly【26†L99-L107】 – you should improve that by hashing the stored password. Also, add a delay or captcha after successive failed attempts to thwart brute force.

- **Session Management Post-Login**: After successful login, besides session renewal, you might want to store an object like `UserCredential` in the session (as in ZK Essentials examples【26†L75-L83】【26†L106-L114】). Make sure this object only contains necessary info (user id, roles, name). Avoid storing the password or other sensitive tokens in session. This session attribute can be checked in each page or composer to verify the user is logged in.

Page Access Checks: Implement a simple check in page initializations: e.g., in each secured ZUL's controller:

java
CopyEdit
```java
if (Sessions.getCurrent(false) == null ||
Sessions.getCurrent().getAttribute("userCredential") == null) {
    Executions.sendRedirect("/login.zul");
    return; // stop further init
}
```

- This ensures that if someone tries to load the page without a login, they get redirected. It's a backstop if someone found a direct URL. In a larger app, a filter or a base composer class that all secure pages extend can do this globally.

- **Logout**: Provide a logout button that calls `Sessions.getCurrent().invalidate()` or uses Spring Security's logout if integrated. After invalidation, ZK will detach the desktop. Redirect the user to a public page (login or home). Also, clear any client-side tokens or identifiers if you set any.

- **Audit Login Events**: As mentioned, log successful and failed logins (but not
  zkoss.org

  zkoss.org
  se logs for suspicious activity.

Security Tips for ZK Components and Data Exposure

Finally, here are additional tips for safe usage of specific ZK components and patterns that often come up in advanced ZK development:

- **ZK Components Executing Scripts**: Use them carefully.
  - The `<zscript>` component is powerful but consider alternative designs. For maintainability and security, it's often better to put Java logic in a **Composer (MVC)** or **ViewModel (MVVM)** class, rather than inline in ZUL. This way, your code is in Java (statically compiled, checked) and less prone to injection mistakes. `<zscript>` is best used for very small glue logic or legacy code. If you do use `<zscript>`, restrict it to Java language (the default) when possible, and avoid dynamic evals as described earlier.
  - If you need client-side scripting, ZK offers `<client>` annotations or `<attribute>` with `client*` to attach client-side listeners, etc. Those are safer than constructing raw script. Use ZK's **JavaScript APIs** or Angular-style data binding rather than injecting script strings. For example, use `onClick="someFunction()"` that's declared in a script file, rather than building a script with user content.
  - **Clients.evalJavaScript** – we reiterate: this should be a last resort. Before using it, ask if ZK has a built-in way to achieve the effect. Often, ZK components/attributes or the `AuResponse` mechanism can do complex tasks without manual JS. Minimizing direct JS reduces XSS risk.
- **Components that Expose Data**:
  - ZK abstracts the client-server communication, but remember that certain data must go to the client to render the UI. For example, if you have a listbox with 100 items, those items' data might be sent to the browser (unless you use paging or on-demand loading). If those items contain sensitive info that the user shouldn't have yet, don't load them all. Load only what's needed for display, or mask

certain fields until the user takes an action that is authorized.

  ○ **ListModel and Model objects**: If you bind a component to a model (say a `ListModelList`), all data in that model could be exposed to the

Security Issues in ZKoss (ZUL) Applications: Identification and Mitigation

Introduction

ZKoss (ZK) is a rich Java-based web framework that uses **ZUL (ZK User Language)** files for UI definition. While ZK simplifies building UIs, developers must be vigilant about security. This report covers common vulnerabilities specific to ZUL files and advanced backend practices to secure ZK applications. We focus on identifying issues like script injection, cross-site scripting (XSS), and unsafe use of expressions, along with session/login security. Real-world ZUL examples, detection steps, and mitigation strategies are provided for each case.

Common Vulnerabilities in ZUL Files

1. Script Injection Vulnerabilities (`<zscript>` Blocks)

ZUL files allow embedding server-side scripts using the `<zscript>&#8203;:contentReference[oaicite:39]{index=39}default, ZK uses Java for `<zscript>`, but it can run other languages (Groovy, JavaScript, Ruby, Python) via the `language`attribute【8†L49-L57】. **Script injection** occurs when an application executes dynamically constructed code that includes untrusted input, leading to **remote code execution (RCE)**. Groovy scripts in`<zscript>` are particularly sensitive because Groovy is dynamic and powerful.

Example Vulnerable ZUL (Groovy Injection):

xml
CopyEdit
```
<window title="Admin Console">
  <zscript language="Groovy">
    // BAD: Evaluating user-provided script (highly insecure)
    def userScript = Executions.getCurrent().getParameter("code");
    if (userScript != null) {
        new GroovyShell().evaluate(userScript);  // Vulnerable:
executes arbitrary Groovy code
    }
  </zscript>
</window>
```

- In
  zkoss.org

  zkoss.org
  er who passes a malicious `code` parameter (e.g., `System.exit(0)` or spawning a shell) can execute arbitrary server-side code. Using `GroovyShell.evaluate()` on a dynamic string is a known code injection sink【4†L205-L212】【22†L3204-L3212】.

- **Identification (Detection Steps):**

  - **Search ZUL files for `<zscript>` usage** – Review all occurrences of `<zscript>` and note the `language`. Pay extra attention to `language="Groovy"` or other scripting languages. These indicate places where server-side code is embedded.

  - **Examine for dynamic code execution** – Within each `<zscript>` block, look for use of classes like `GroovyShell`, `GroovyClassLoader`, or eval/execute methods. Also flag any use of `Executions.create*` methods that compile or run dynamic content. For Java `<zscript>`, check for use of reflection or `Executions.evaluate()` (if any).

  - **Trace input sources** – If the script constructs code from variables or parameters, determine if any of those come from user input (e.g., `Executions.getCurrent().getParameter`, component values, or session attributes set by users). Any concatenation of user data into code is a red flag.

  - **Real-world clues** – Often, script injection issues arise in admin or customization features (e.g., allowing admins to input scripts for custom logic). Identify any such feature in documentation or code comments and review it thoroughly.

- **Mitigation Strategies:**

  - **Avoid Dynamic Evaluation:** The safest approach is not to evaluate user-supplied code at all. Implement required logic in Java (controllers or view models) rather than via dynamic Groovy scripts. This keeps code static and compiled, eliminating this injection vector.

- **Restrict Scripting Capabilities:** If dynamic scripts are absolutely needed (e.g. plugin systems), do not expose them to untrusted users. Require strong authentication and limit this feature to admin roles. Additionally, sandbox the script environment: disable dangerous classes/methods. For example, configure a custom Groovy `ClassLoader` or use Groovy's secure features to whitelist permitted operations.

- **Input Validation:** Rigorously validate or sanitize any input that might be used in a script. *Never* pass raw user input to `GroovyShell.evaluate()` or similar. If you must accept expressions (say, a math formula), parse and validate the format before evaluation.

- **Use Static Analysis Tools:** Employ tools which have detectors for Groovy script injection. For instance, FindSecBugs has rules to catch usage of GroovyShell with dynamic input【4†L205-L212】. This can help flag vulnerable code during development.

- **Least Privilege:** Run the application with minimal privileges. Even if code injection occurs, a locked-down security manager or low-privileged OS account can limit damage. (Advanced setups might remove scripting engine libraries entirely from production if not used.)

By treating any `<zscript>` as executable code and avoiding inclusion of unvalidated data, you prevent attackers from injecting scripts that the server would run.

2. Cross-Site Scripting (XSS) in ZUL Content

Cross-site scripting is a pervasive web vulnerability where malicious scripts are injected into pages viewed by others【1†L56-L64】. In ZK, XSS typically arises when **user input is included in ZUL views without proper encoding**. The ZK framework encodes many outputs by default (to help developers), but there are exceptions where you must be cautious.

How ZK Handles Encoding: ZK automatically escapes special characters (`<`, `>`, `&`, etc.) for all standard input components and label outputs. For example, using a value binding in a textbox or label is safe:

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

- Even if `any_value` contains `<script>alert('XSS')</script>`, ZK will escape it (the `<` becomes `&lt;`, etc.)【1†L68-L76】. In fact, an EL expression in a ZUL page implicitly renders a Label, which is encoded【1†L81-L84】. **This means normal data binding to UI components is generally safe from XSS by default.**

- **Unsafe Cases (When ZK Does *Not* Encode):** Certain ZK components and techniques deliberately allow raw HTML for flexibility. These will **not auto-escape** content and thus can introduce XSS if misused:

`<html>` Component (Html.setContent) – Intended to embed raw HTML. Content is not encoded【29†L127-L135】. Example:

xml
CopyEdit
```
<html>${userComment}</html>
```

  - If `userComment` includes a script tag, it will be rendered as-is, executing malicious script. (Note: Since ZK 10.0.0, the Html component sanitizes content by default to reduce XSS risk, but developers are advised *not* to rely solely on this and never include raw `<script>` in user-provided content【2†L103-L107】.)

  - **Comboitem Content** – The `<comboitem>` component's `content` attribute is raw HTML and not encoded【29†L127-L135】. Any user-supplied HTML here can execute scripts.

  - **Native HTML in ZUL** – Embedding native HTML tags directly in ZUL (using the `native` or `xhtml` namespace) bypasses ZK's encoding【1†L87-L95】. For example, `<div xmlns="native">${data}</div>` will inject raw HTML from `${data}`.

Client-side Utility Methods – Using `Clients.showNotification(String)`, `Clients.alert(String)`, or `Clients.evalJavaScript(String)`. These do **not** encode their input by design【2†L113-L120】. They allow formatting or direct script execution on the client. If you pass user data directly, it can inject scripts. For example:

java
CopyEdit
```
// In a ZUL event listener or ViewModel (Java code)
String name = userService.getName();          // could be
attacker-controlled
```

```
Clients.evalJavaScript("showWelcome('" + name + "')");
```

- If `name` contains a quote or `</script>`, it could break out of the string and run arbitrary JS. Similarly, `Clients.showNotification("Success: " + userInput)` will render any HTML in `userInput`【2†L113-L120】. Developers must manually escape or sanitize data in these cases.

- **Page Directive Attributes** – All attributes of the ZUL `<?page?>` directive are output without encoding【2†L129-L137】. In practice these are usually static (set by developers), but if any page attribute were set from user input, it could be a vector.

Example Vulnerable ZUL (XSS):

xml
CopyEdit
```
<window title="Feedback">
  <!-- Developer allows some HTML in userComment -->
  <html>${param.userComment}</html>
  <button label="Preview"
          onClick='Clients.showNotification(param.userComment,
"info")'/>
</window>
```

- If `param.userComment` is `"<img src=x onerror=alert(1)>"`, the `<html>` component will inject an actual image tag with an onerror script, triggering an alert. The `Clients.showNotification` will also display it unsanitized, which could execute the script or break the UI【2†L113-L120】.

- **Identification (Detection Steps):**

  - **Find raw HTML uses** – Grep for `<html>` components or `.setContent(` calls in Java. These are likely points where raw output is used. Check if the content comes from user input (e.g., form fields or database content that users can influence).

  - **Check comboitem and native tags** – Search for `comboitem content=` in ZULs. Also look for usage of `xmlns:native` or `xmlns:xhtml` in ZUL files which signals direct HTML embedding.

- *Scan for Clients. calls** – In ZUL onClick/onChange handlers or controllers, find calls to `Clients.evalJavaScript`, `Clients.showNotification`, `Clients.alert`, etc. **Audit what data is being passed**. If any argument originates from a user (request param, component value, etc.) and isn't encoded or sanitized, that's a potential XSS sink.

- **Identify disabled encoding** – Ensure no code explicitly disables encoding on components. E.g., calling `Label.setRawValue()` (if used) or setting a label's `encode=false` (in older ZK, Label had a flag to allow HTML). Such patterns should be treated as dangerous unless the content is trusted.

- **Dynamic testing** – Run the app and input typical XSS payloads (e.g., `<script>alert(1)</script>` or `"><svg onload=alert(1)>` in form fields) to see if they appear unescaped in the UI. This helps catch any XSS issues that static review might miss.

- **Mitigation Strategies:**

  - **Prefer Safe Components:** Use ZK's normal components (Label, Textbox, etc.) for user-generated content, as they encode dangerous characters by default【29†L117-L125】. For example, display user comments in a `<label value="${comment}"/>` rather than an `<html>` element. If multiline text is needed, use `<label multiline="true">` (which still escapes HTML) or a readonly Textbox.

  - **Sanitize if Raw HTML Needed:** If your use-case requires allowing some HTML (e.g., bold/italic in user input), sanitize it on the server before rendering. Use libraries like **OWASP Java HTML Sanitizer** or **Jsoup** to whitelist tags and remove scripts. ZK's utility `XMLs.escapeXML(String)` can escape text【2†L133-L140】, though that will strip **all** HTML. For partial HTML, a sanitizer that allows certain tags is better. Always strip `<script>` tags or `on*` event attributes entirely.

  - **No Inline Scripts:** Never allow raw `<script>` tags via user input. Even with ZK 10+ auto-sanitizing `<html>` content, do not rely on it completely【2†L103-L107】 – explicitly remove or disallow script content.

Escape Data in Client Calls: For methods like `Clients.evalJavaScript`, avoid building JavaScript by string concatenation. If you need to pass user data to a client-side function, use JSON encoding or ZK's JSON utilities. For example:

java
CopyEdit
```
String safeName = org.zkoss.json.JSONValue.escape(name);
Clients.evalJavaScript("showWelcome('" + safeName + "')");
```

- This ensures special characters in `name` are escaped in the JavaScript context. Similarly, for `showNotification`, if you want to display user text, use `Strings.escape(userInput)` (from ZK utils) or send it as a second parameter indicating content should be treated as text.

- **Be Aware of Encoding Gaps:** Remember which components/properties do not auto-encode【29†L127-L135】. Avoid binding user data to them unless you sanitize first. If you use `Html.setContent()` or `<comboitem content=...>` for user data, you are responsible for cleaning it【29†L129-L137】. Incorporate these rules in code reviews or create a checklist for ZUL code.

- **Content Security Policy (CSP):** (Advanced) Consider using a strict CSP header to mitigate the impact of any XSS that might occur. For example, disallow inline scripts and only allow scripts from your domain. This won't fix XSS, but it can make exploits harder (e.g., preventing injected scripts from executing or loading external resources).

By encoding or sanitizing all user-supplied content, you can prevent XSS in ZK applications. ZK's default behavior is safe for standard use; issues arise when developers bypass those safeties for flexibility, so weigh that need carefully against security risks.

3. Improper Use of EL Expressions and Dynamic Content Rendering

ZK supports powerful expression language (EL) binding and dynamic UI composition. **Improper use of these features can introduce security issues** such as unauthorized page inclusion, template injection, or data exposure. This category covers scenarios like including the wrong ZUL page based on user input and unsafe use of ZK's dynamic UI creation.

Unsafe Dynamic Page Inclusion (ZUL Path Injection): Developers sometimes compose pages dynamically. For instance, using a parameter to decide which ZUL to include:

xml
CopyEdit
```
<!-- Vulnerable if 'page' param is not validated -->
<include src="/views/${param.page}.zul"/>
```

- If `param.page` is attacker-controlled, this can load arbitrary ZUL files. An attacker might access pages they shouldn't (e.g., `admin.zul` or other users' pages). While direct access to `/WEB-INF` is blocked by the servlet container, including a page might still expose sensitive UIs or at least reveal their existence. **Treat any user input that influences file paths as dangerous** (akin to directory traversal vulnerabilities).

Template Injection via `Executions.createComponents`: ZK's API allows creation of components from ZUL content at runtime:

java
CopyEdit
```java
String zulPath = Executions.getCurrent().getParameter("zulPage");
Executions.createComponents(zulPath, parentComponent, null);
```

- If `zulPath` is user-supplied (from a query param, etc.), an attacker could try to load unintended zul files. Even more dangerous is `Executions.createComponentsDirectly(String zulMarkup, ...)` which evaluates a Zul markup string. Passing user input to that is equivalent to letting users execute code or HTML in your app. This is a serious risk – essentially remote template injection leading to XSS or worse.

- **Improper EL Usage:** ZK's EL in data binding is generally safe because it's defined by the developer, not the user. However, misuse can occur:
    - If you design something where user input is fed into an EL evaluation (e.g., letting users specify an EL expression for a field binding), this could be abused to call unintended methods. While ZK doesn't provide a straightforward way for users to supply EL, be cautious of any feature that might eval EL from strings.
    - Another aspect is accidentally exposing server data via EL. For example, ZK EL might allow accessing static resources or system properties if you bind them. `${systemProperties['os.name']}` would display the server OS name. That's not an "attack" per se, but leaking such info could aid an attacker. Review your ZUL bindings to ensure you're not exposing sensitive server information.
    - Ensure that data-binding expressions do not call methods with side effects unless intended. In MVVM, methods annotated with `@Command` or `@GlobalCommand` should include security checks as needed, since binding can trigger them based on UI actions.

Example Issue (Dynamic Include):

xml
CopyEdit
```
<window apply="com.app.MyComposer">
  <!-- Show different page based on 'view' request parameter -->
  <zscript>
     String page = Executions.getCurrent().getParameter("view");
     if(page != null) {
         Include inc = new Include("/zul/" + page + ".zul");
         self.appendChild(inc);
     }
  </zscript>
</window>
```

- If an attacker accesses `?view=adminDashboard`, this code will load `adminDashboard.zul` even if the user isn't an admin. Or `?view=../config` might attempt to load `../config.zul` (likely an invalid page, but it could expose an error or be manipulated to traverse directories). This is both an injection and an access control problem – the user can force the application to include pages out of the intended flow.

- **Identification (Detection Steps):**

  - **Audit includes and navigation logic** – Search for `<include src="${...}">` in ZUL or any usage of the `<include>` component with variable input. Also search Java code for `Executions.createComponents(` or `Include` object creation. Wherever a path is built dynamically, flag it.

  - **Trace inputs for dynamic paths** – If a dynamic include or component creation uses a variable, find how that variable is set. Is it from `Executions.getCurrent().getParameter()`? from a cookie? from user preferences? If it's user-controlled and not validated against an allowlist, it's a potential vulnerability.

  - **Check for direct ZUL eval** – Look for any use of `Executions.createComponentsDirectly` or similar functions that take raw ZUL content. These should never be fed with unsanitized external input. Also, verify that any custom UI templating you do (if any) doesn't allow injection.

  - **Review EL bindings** – Scan through ZUL files for unusual EL expressions. If you see something like accessing system properties, static classes, or executing expressions that could divulge data, note them. For instance, if a Label value is bound to something like `${vm.secretData}`, ensure that `secretData` is not something that should stay on the server. Usually, the presence of sensitive info in the view model implies it will go to the client. If it's not needed on the client, don't bind it.

- **Test parameter manipulation** – Try altering any URL parameters or UI state that selects templates. If your app has a "page" or "view" parameter, attempt to load pages you shouldn't by changing it. If you find you can access an admin page while not logged in, that's a major issue to fix (via proper access control, see mitigations below).

- **Mitigation Strategies:**

Validate and Whitelist: Never use raw user input to select a ZUL page or template. If you need to switch views based on input, maintain a **whitelist** (mapping) of allowed pages. For example:

java
CopyEdit
```java
Map<String, String> pageMap = new HashMap<>();
pageMap.put("home", "/zul/home.zul");
pageMap.put("dashboard", "/zul/dashboard.zul");
...
String view = Executions.getCurrent().getParameter("view");
if (pageMap.containsKey(view)) {
    Include inc = new Include(pageMap.get(view));
    parent.appendChild(inc);
} else {
    // Invalid page request – handle error or redirect
}
```

- This ensures only known pages load. Any attempt to supply a different value is ignored or handled as an error.

- **Secure ZUL Locations:** Keep sensitive ZUL files out of public reach. Place admin or internal pages under `WEB-INF` (so they can only be included programmatically, not via direct URL)【12†L79-L84】, or use ZK's authentication to guard them. Even if an attacker knows the path, they shouldn't be able to load it without going through your application's security checks.

- **Use Roles/Permissions in UI flow:** Instead of letting the client decide which page to load, make such decisions on the server with knowledge of the user's role. For example, if `view=adminDashboard` comes in, check on the server if the current user is an admin. If not, refuse or redirect. This ties into session/login security (broken access control prevention).

  - **Avoid User-Supplied Templates:** Do not allow users to provide raw ZUL or template strings that get rendered. If you have a use-case like user-customizable dashboards or rich text editing, sanitize that input thoroughly or use a safe format (for rich text, use a markup like Markdown that you can safely convert to HTML server-side). Treat any such functionality as you would an HTML editor – with strict sanitization on save and on display.

  - **Limit EL Capabilities:** By default, ZK's EL is restricted to property access and method calls on allowed objects (view model, composer, etc.). Avoid customizations that would broaden that. For instance, do not globally expose something like `System.getProperties()` to EL context. If you use ZK Functions or static method bindings, ensure they don't expose internals. It's generally safe out of the box, but be mindful when extending ZK.

  - **Stay Updated:** Use the latest ZK version to benefit from security improvements. ZK 10 and above include enhancements like stricter URI handling (preventing some inclusion attacks)【10†L93-L96】 and the Inaccessible Widget Blocker (discussed later) for access control. Also, patch known vulnerabilities: e.g., a recent vulnerability (CVE-2022-36537) in the AuUploader servlet allowed unauthorized file access via a crafted request【5†L123-L131】 – upgrading ZK patched this. Keeping up-to-date protects you from such edge-case exploits in dynamic content handling.

By reducing overly dynamic behavior and tightly controlling what content and pages can be rendered, you eliminate an entire class of injection and inclusion problems. Essentially, **never let the user decide what code or page executes** – always mediate and validate their requests on the server.

Backend Java Security Practices for ZK Applications

While the focus is on ZUL files, the backend Java code in a ZK app plays a critical role in security. Below are practices (beyond the basics) that experienced developers should follow:

- **Server-side Input Validation & Injection Prevention:** ZK is mostly a UI framework; it won't automatically secure your database or backend integrations【12†L66-L74】. Apply standard injection prevention in your Java code:

- Use **parameterized queries or ORM safety** for database access (to prevent SQL injection). Never construct SQL by concatenating user input【12†L69-L72】.
- For LDAP queries, OS commands, or other interpreters, ensure inputs are sanitized or use safe APIs (e.g., prepared statements for LDAP, `ProcessBuilder` for commands with arguments).
- Validate input lengths, ranges, and patterns. ZK's component constraints can handle some validation (e.g., `<textbox constraint="/^[A-Z0-9]{6}$/" />` for a code format), but always re-validate on the server side, as clients can bypass UI constraints.
- Be cautious of **insecure deserialization**. If you use Java serialization (e.g., storing objects in session or caching user data to disk), use allowed-classes lists or avoid serialization of user-influenced objects. Prefer JSON/XML for data interchange and validate that as well.

- **Sensitive Data Handling:** Only send or store what is necessary:
  - **Least Exposure to Client:** Avoid sending sensitive data to the client unless absolutely needed. For example, don't embed a user's full SSN or password in the UI (even if hidden). If you have a form that needs such data, consider providing it only when the user requests or needs to view it, rather than on initial page load.
  - **Storage in Session:** Store minimal information in the user's session. A user ID and roles/permissions are usually enough. Don't keep things like plaintext passwords, credit card numbers, or large objects in the session. Not only is it a security risk if session data leaks, but it also affects performance.
  - **Protect Files and Resources:** If your ZK app works with files (exports, reports, images), keep them in secure locations. For example, if users upload files, save them in a directory not directly accessible via URL, and stream them via a controlled servlet when needed. Similarly, if your app reads configuration or user data files, ensure they are in non-web-accessible paths and with proper ACLs on the filesystem.
  - **Masking and Encryption:** If certain data must be stored or transmitted, use encryption or masking. For instance, encrypt sensitive fields in the database. Within the UI, mask secrets (e.g., show API keys as `****1234` with an option to reveal on demand). This way, even if the UI is compromised, not everything is exposed.

- **Access Control (Enforce on Server):** Don’t rely solely on the client-side to hide or disable protected features:
    - Implement robust authentication and authorization. ZK itself doesn’t provide a login module【12†L75-L79】, so integrate something like **Spring Security** or container-managed security. These frameworks handle password hashing, login attempts, and can restrict URLs or method calls by roles.
    - On every server-side event that performs a sensitive action, check the user’s permissions. For example, if there’s an `onClick` event for an “Delete User” button bound to a method `deleteUser(userId)`, that method should verify the current user is an admin or owns that `userId` before proceeding.
    - Use **InaccessibleWidgetBlockService** (IWBS) for defense in depth. Since ZK 10.0.0, IWBS is on by default, blocking events from UI components that are not supposed to be interactive (like disabled or not rendered to the client)【10†L69-L77】. This helps prevent a malicious user from enabling a hidden button via browser dev tools and clicking it. If you’re on an older ZK version, you can enable this by configuring a listener in `WEB-INF/zk.xml` or upgrading to 10+. IWBS complements your own checks by automatically rejecting illegal UI interactions at the framework level.
    - **Broken Access Control Testing:** Ensure that pages or functionalities meant for certain roles cannot be accessed by others. This might mean trying to load admin pages as a normal user (should be prevented via redirect or error) and ensuring UI components meant for admins truly don’t do anything if somehow triggered by a non-admin.
- **Logging and Monitoring:** Implement logging for security-relevant events in your ZK application:
    - Log authentication events (login success/failure, logout).
    - Log access control failures (e.g., if a user tries to perform an action and is denied, record it).
    - Capture unusual input in logs (e.g., catch exceptions that might indicate attempted injection or misuse, and log the input that caused it).
    - Use these logs. Monitor them or feed them into a SIEM (Security Information and Event Management) system. This is important for detecting attacks in progress or identifying weaknesses. For example, multiple XSS attempts in logs could indicate a probing attacker, prompting a code review in that area.

- **Security Configuration:** Leverage server and framework settings:
  - Ensure you have a **secure deployment configuration** – e.g., disable directory listings on the web server, use HTTPS, set secure cookies, and if using a cluster, secure the communication between nodes.
  - If your application is large, consider a Web Application Firewall (WAF) which can provide generic protections (though be cautious as AJAX applications like ZK might not work with all WAF rules without tuning).
  - Set appropriate **HTTP headers** for responses from your ZK app: `X-Frame-Options: SAMEORIGIN` to prevent clickjacking, `X-Content-Type-Options: nosniff`, `Referrer-Policy`, etc. These can usually be added via a servlet filter or proxy server.

In summary, treat your ZK application as you would any enterprise web app: follow OWASP Top 10 practices (ZK's documentation even maps these risks to ZK specifics【10†L43-L52】) and never assume the framework alone has you covered. ZK provides many hooks and default protections, but how you use it determines the security outcome.

Session and Login Security in ZKoss Applications

Session management and authentication are fundamental to ZK application security. Here's how to address them in the context of ZK:

Session Security in ZK

ZK applications are stateful, relying on the server session to store UI state and user data. Each client browser window is associated with a ZK **Desktop** (an instance representing the page view on the server) identified by a unique desktop ID token. Maintaining session security involves both standard web practices and understanding ZK's mechanism:

- **CSRF Protection via Desktop ID:** By design, ZK is an Ajax-driven framework with no reliance on static form submissions, which inherently reduces CSRF risk【16†L83-L92】. Every ZK Ajax request includes a unique desktop ID (a GUID) that was generated when the page was loaded【16†L89-L98】【16†L109-L117】. The server knows the valid desktop ID for that session and page, and will reject requests with missing or incorrect IDs as invalid. This acts like a CSRF token – an attacker would have to guess a valid, ephemeral ID for the target's session, which is impractical. Additionally, component IDs and event references must match the server's expectations, or the request is ignored【16†L89-L98】. **Bottom line:** ZK's architecture gives a layer of CSRF defense automatically.

    - *Tip:* You can enhance this by also checking the `Origin` or `Referer` header in a servlet filter for sensitive actions (defense-in-depth to ensure the request comes from your domain)【16†L72-L80】. But this is often not necessary unless you have non-ZK endpoints.
    - For traditional multi-page flows or file download links, do implement CSRF tokens in those forms since they might not use the ZK AJAX engine.

- **Session Hijacking Prevention:**

Session Fixation: Always invalidate the old session upon successful login to get a new session ID. If using Spring Security, this happens automatically (it creates a new session and transfers attributes)【25†L1-L4】. For custom logins, explicitly do:

java
CopyEdit
```
Sessions.getCurrent().invalidate(); // invalidate ZK session
(underlying HttpSession)
Sessions.getCurrent(true);         // create a new session
```

    - (If using `Executions.sendRedirect()` to post-login page, you might let the container create a new session on redirect.) The idea is to ensure an attacker cannot plant a session ID (via an insecure pre-login link) and have the user adopt it.

Secure Cookies: Configure your web container or via code to mark the JSESSIONID cookie as HttpOnly and Secure. HttpOnly prevents JavaScript from accessing the cookie (mitigating XSS stealing the session), and Secure ensures it's only sent over HTTPS. This is typically done in `web.xml` with:

xml
CopyEdit
```
<session-config>
  <cookie-config>
    <http-only>true</http-only>
    <secure>true</secure>
  </cookie-config>
  <session-timeout>30</session-timeout> <!-- timeout in minutes -->
</session-config>
```

    -

    - **Session Timeout:** Set a reasonable session timeout (e.g., 15-30 minutes of inactivity, depending on your app). ZK will automatically clean up desktops on session expiration. A shorter timeout limits the window for hijacking if someone walks away from a logged-in session.

    - **Concurrent Sessions & Logout:** Decide if users can have multiple sessions (e.g., login from multiple devices/browsers). If not, enforce single session per user (Spring Security can control this). Also, ensure that on logout, you invalidate the session (`Sessions.getCurrent().invalidate()`) and perhaps also remove any client-side tokens or data. Encourage users to log out especially on shared computers.

    - **Monitor Session for Anomalies:** In high-security contexts, you might implement session-IP binding (if IP changes mid-session drastically, flag it) or device fingerprinting. ZK doesn't do this out-of-the-box, but you can get client info via `Executions.getCurrent().getRemoteAddr()` and user-agent. Be cautious: legitimate reasons (mobile network, corporate proxies) can cause IP to change.

- **Resource Cleanup:** ZK sessions (desktops) hold a lot of state. Ensure you do proper cleanup to avoid leaking data:

    - If you store custom objects in session or desktop attributes, remove them on logout or when no longer needed.

    - Use ZK's `SessionCleanup` or `ExecutionCleanup` listeners if needed to clean sensitive info after requests.

    - If using server-push or background threads, be careful not to keep references to session-bound data longer than necessary.

Login Security in ZK Applications

Authentication is often custom in ZK, but you can integrate established frameworks:

- **Use Established Security Frameworks:** Leverage **Spring Security** with ZK for a robust solution【23†L8-L12】. There are ZK tutorials and examples for this integration. Spring Security will handle password encoding, login, logout, remember-me, session fixation, CSRF (for non-Ajax), and even UI-level method security (like `@PreAuthorize` annotations in your view model methods).

    - If using Spring Security, note that ZK events will still go through, but you can secure them by checking roles in methods or using Spring Security's annotation on methods invoked by ZK (e.g., a service called in a ZUL event).

  - Spring Security can also secure the ZUL page URLs. For example, you can configure `/admin/**` URL pattern to require ROLE_ADMIN, which means if an attacker tries to load an admin ZUL, they get redirected to login automatically.

- **Custom Login Implementation:** If not using a framework, do it carefully:

  - **Password Handling:** Always store passwords hashed (e.g., BCrypt). On login, hash the input password and compare with stored hash. Never store or log plaintext passwords. Use a strong hashing algorithm – this is critical if your database leaks.

  - **Login Form Security:** Use `<textbox type="password"/>` for password fields so it's masked. ZK will treat it as a normal input; ensure the form is submitted over TLS. You might have a login button that triggers an event to verify credentials. On failure, give a generic error ("invalid login") to avoid username enumeration.

  - **Brute Force Mitigation:** Implement a counter for login attempts. For example, after 5 failed tries for a user account, lock it for 15 minutes. Or use a CAPTCHA on the login form after a few failures. This can be implemented in your auth service logic.

  - **Logging:** As mentioned, log login attempts (especially failures with the username, timestamp, and source IP). This helps detect brute force or credential stuffing attacks.

  - **Post-Login:** After login, set a session attribute like `session.setAttribute("user", userObject)` to mark the user as authenticated. Use this in access checks on pages (as shown in ZK Essentials: checking a `UserCredential` in session)【26†L61-L69】【26†L75-L83】. Then redirect to the main page. (Remember to invalidate the previous session if you haven't yet.)

  - **Logout:** Provide a logout function that calls session invalidate. In ZK, you might have a "Logout" button that does `Sessions.getCurrent().invalidate(); Executions.sendRedirect("/login.zul");`. Once invalidated, the ZK desktop is destroyed and any further action requires a new login. Also clear any client state if you stored something in cookies (for example, if you implemented a "remember me" cookie yourself, remove it).

- ◦ **Session Transfer on HTTPS Downgrade:** Ensure the app doesn't allow switching to HTTP after login. If a user logged in over HTTPS, they should not continue the session over HTTP (or the cookie could be intercepted). Ideally force all traffic over HTTPS.

- **Protecting Pages and Components:** Even with login done, ensure that each page verifies the user's role:

Use a filter or a check at page load (composer or view model initialization) to redirect unauthorized users. For instance, in an admin page's controller, do:

java
CopyEdit
```java
if (!user.isAdmin()) {
   Executions.sendRedirect("/forbidden.zul");
   return; // stop further setup
}
```

  - This prevents an authenticated but non-admin user from seeing admin content if they somehow reach the page.
  - On the client side, you might hide menu options for unauthorized users, but remember those are just cosmetic. The real enforcement is on the server when the page or action is attempted.

- **Two-Factor Authentication (2FA):** For advanced security, consider adding 2FA for login. This could be integrated via Spring Security 2FA modules or custom code (sending an OTP via SMS/Email and prompting the user). If implemented, treat the OTP input like any sensitive input (validate it, rate-limit attempts, etc.). ZK can easily create a UI for OTP entry.

- **Error Handling:** Be mindful of error messages during login. If using ZK's `Messagebox` to show errors, that's fine – just don't reveal, for example, "User exists but password is wrong" vs "User not found". Keep it generic to not give away whether a username is valid.

Security Tips for ZK Components and Data Exposure

Finally, here are additional tips for safe usage of specific ZK components and patterns that often arise in advanced ZK development:

- **Use `<zscript>` Sparingly:** We covered script injection risks – in general, keep business logic out of ZUL `<zscript>` blocks. Use Java composers or view models to handle logic and call secure backend services. This not only improves security (less inline code to audit) but also maintainability.

- **Component Visibility vs. Existence:** To secure UIs, prefer not to render components at all rather than hiding them. For example, use `unless` attribute to conditionally create a component only for authorized users, instead of creating it and setting `visible="false"`【27†L39-L47】【27†L43-L50】. If it's never rendered to the client, an attacker can't force it visible. If you must create and hide (maybe for dynamic UI reasons), then rely on IWBS and server checks to prevent misuse.

- **Components that Hold Data:** Some components can store data that might not be obviously visible:

  - **Listbox/ListModel:** If you use a `ListModelList` or other models, by default ZK only renders the needed items to the client. However, if the list is small or not using paging, a user could view source or use ZK's client-side widget API to see list items' labels. Ensure that any sensitive info in list items (like an email or ID) is something the user is allowed to see. If not, don't include it in the model or mask it.

  - **Component Attributes:** ZK components allow you to set custom attributes (e.g., `comp.setAttribute("secret", value)`). These are server-side only by default and not sent to the client. However, be careful: if you set an attribute and then later use it in a client-side context (like `Clients.showNotification(comp.getAttribute("secret"))`), you might accidentally expose it. Treat server-side stored secrets with care and remove them when done.

  - **Serialization to Client:** Some data may serialize to the client as part of component state. For instance, if you use certain ZK features like client binding or add custom properties to components via `setAttribute` with `Component.COMPONENT_SCOPE`, they might be marshaled to the client. Avoid attaching sensitive data directly to UI components in a way that could go to the browser. When in doubt, test what is visible on the client (ZK's developer tools or browser's dev tools can show you the HTML/JSON being sent).

  - **File Upload/Download Components:** Use ZK's `<upload>` with care. Always validate the uploaded file on the server side (check file type, size, scan for viruses). For file downloads, ensure the user is authorized to get the file. ZK's `Filedownload.save()` makes it easy to send files, but you should verify access before calling it. Keep the repository of files secure (ideally outside webroot or with random file names so they can't be directly URL-accessed).

    - **ZK Charts/Graphs and Data:** If using ZK charts or other components that render data on client side (like Google Maps component, etc.), be mindful that any data you feed them will be on the client. E.g., if a chart has data series values that are sensitive, those values end up in the page source or AJAX responses. Aggregate or anonymize data if needed when sending to client.

- **Desktop/Application Memory:** ZK keeps state on the server, which means memory usage grows with each session. From a security perspective, be aware of **DoS via memory exhaustion**. An attacker opening many sessions or causing large components to be created could try to overwhelm the server. Mitigations:

    - Set `max-desktops-per-session` and `max-requests-per-session` in `zk.xml` to reasonable limits to prevent abuse【29†L141-L149】.

    - Use `AuUploader` and file size limits to prevent huge file uploads from consuming memory.

    - Consider using ZK's server push carefully; limit the number of concurrent pushes or use queuing to avoid floods.

    - Monitor your application's memory and have alerts if it's growing abnormally (could indicate a denial of service attempt or just a memory leak bug).

- **Testing and Updates:** Regularly test your ZK application with security tools. Use OWASP ZAP or Burp Suite to proxy your app and scan for issues like XSS, missing headers, etc. Although automated scanners might not navigate ZK's AJAX easily, you can script some interactions or test specific requests. Also, keep an eye on ZK's release notes for security advisories. For example, if a new ZK version fixes a vulnerability and you're on an older version, assess the risk and update promptly【12†L49-L52】【10†L47-L53】.

By following the above practices and being mindful of how ZK components handle data and execution, you can build a robust, secure ZKoss application. Security is an ongoing process: incorporate these checks into your development and deployment pipeline, and stay educated on new threats and patches. With careful coding and configuration, ZK applications can be made as secure as any traditional web application while providing a rich user experience.