UNIVERSITY OF TRENTO - Italy

Offensive technologies Fall 2017

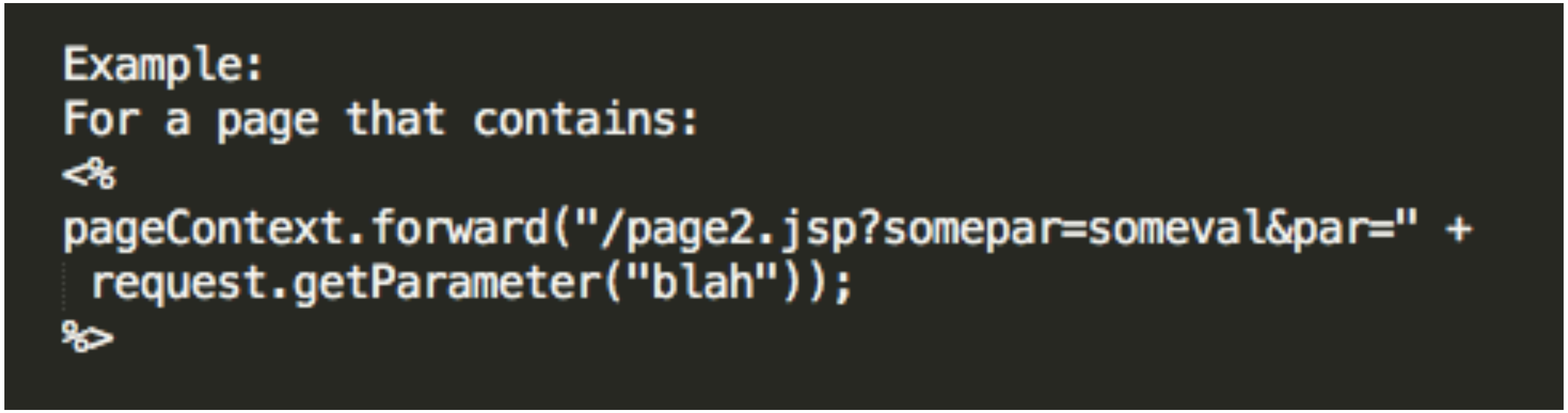
Solutions for the Vulnerability finding Exercise

https://securitylab.disi.unitn.it/doku.php?id=course_on_offensive_technologies

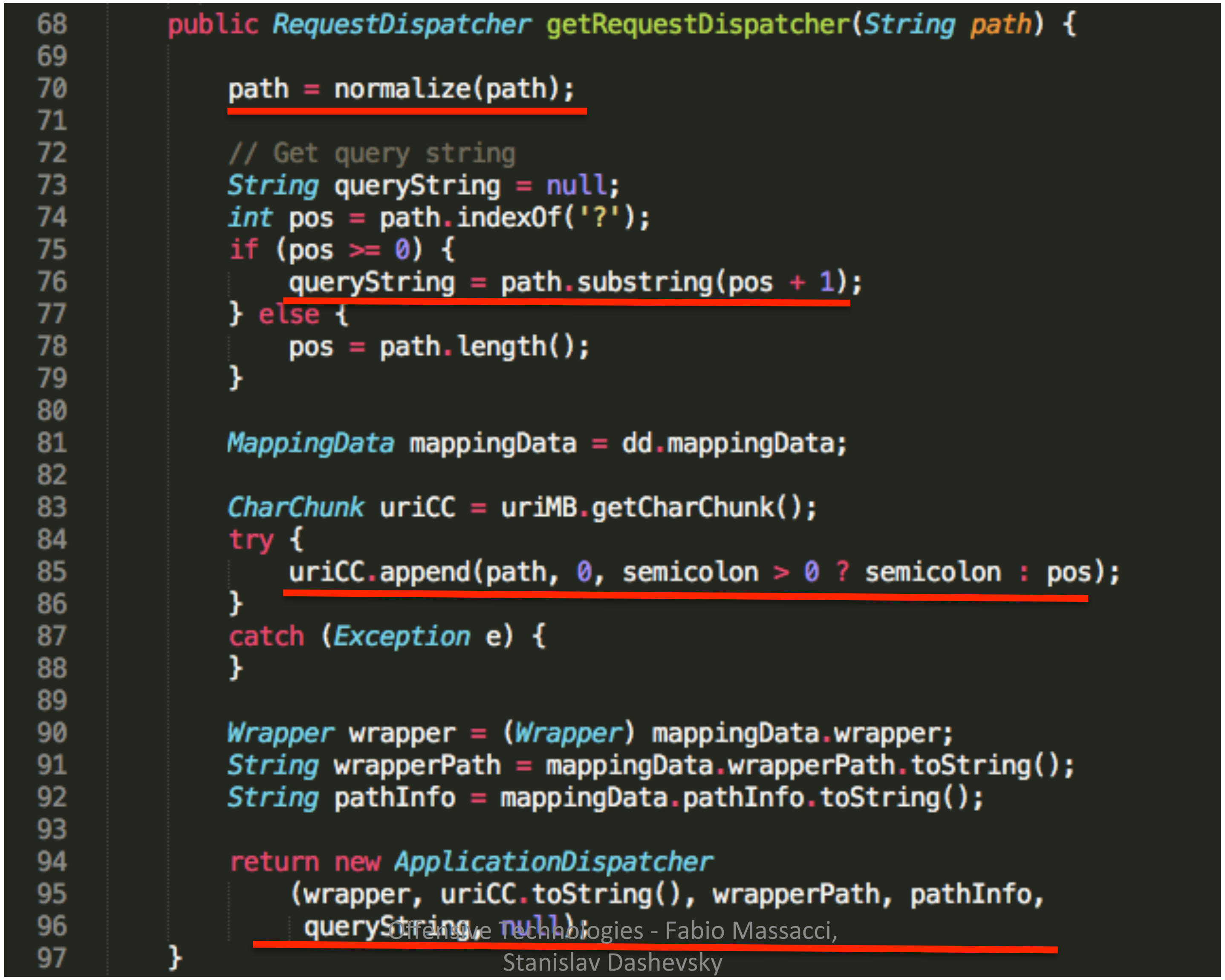
TASK 1 (CVE-2008-2370)

- **Information disclosure (path traversal) in Apache Tomcat**
- When using a **RequestDispatcher** the target path was **normalised** before the query string was removed. A request that included a specially crafted request parameter could be used to access content that would otherwise be protected by a security constraint or by locating it in under the WEB-INF directory.

TASK 1 (CVE-2008-2370)

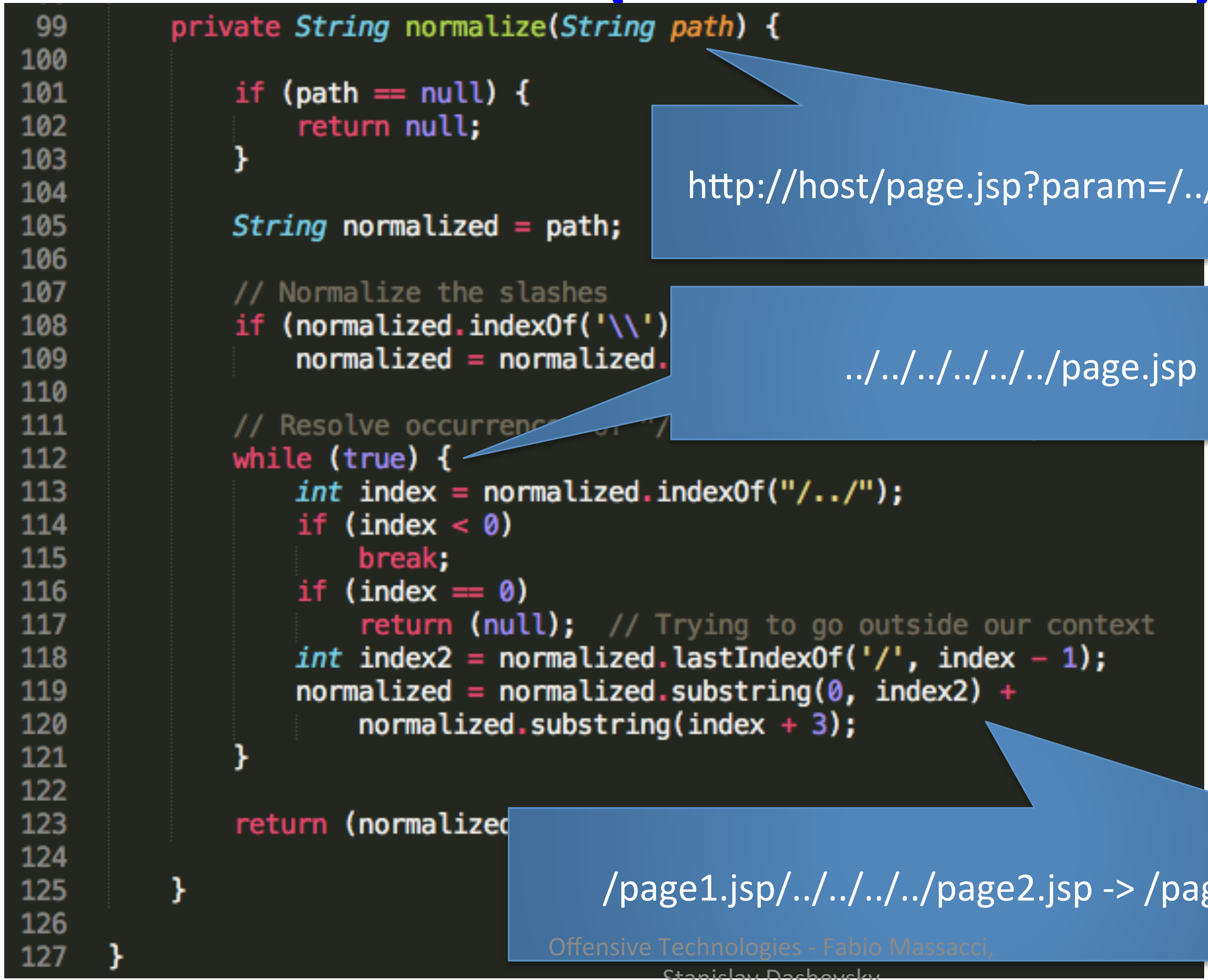

```
Example:
For a page that contains:
<%
pageContext.forward("/page2.jsp?somepar=someval&par=" +
 request.getParameter("blah"));
%>
```

TASK 1 (CVE-2008-2370)



```
    public RequestDispatcher getRequestDispatcher(String path) {

        path = normalize(path);

        // Get query string
        String queryString = null;
        int pos = path.indexOf('?');
        if (pos >= 0) {
            queryString = path.substring(pos + 1);
        } else {
            pos = path.length();
        }

        MappingData mappingData = dd.mappingData;

        CharChunk uriCC = uriMB.getCharChunk();
        try {
            uriCC.append(path, 0, semicolon > 0 ? semicolon : pos);
        }
        catch (Exception e) {
        }

        Wrapper wrapper = (Wrapper) mappingData.wrapper;
        String wrapperPath = mappingData.wrapperPath.toString();
        String pathInfo = mappingData.pathInfo.toString();

        return new ApplicationDispatcher
            (wrapper, uriCC.toString(), wrapperPath, pathInfo,
             queryString, null);
    }
```

TASK 1 (CVE-2008-2370)



```
    private String normalize(String path) {

        if (path == null) {
            return null;
        }

        String normalized = path;

        // Normalize the slashes
        if (normalized.indexOf('\\')
            normalized = normalized.

        // Resolve occurren
        while (true) {
            int index = normalized.indexOf("/../");
            if (index < 0)
                break;
            if (index == 0)
                return (null);  // Trying to go outside our context
            int index2 = normalized.lastIndexOf('/', index - 1);
            normalized = normalized.substring(0, index2) +
                normalized.substring(index + 3);
        }

        return (normalize

    }

}
```

http://host/page.jsp?param=/../WEB-INF/web.xml

../../../../../../page.jsp -> /page.jsp

/page1.jsp/../../../../page2.jsp -> /page2.jsp

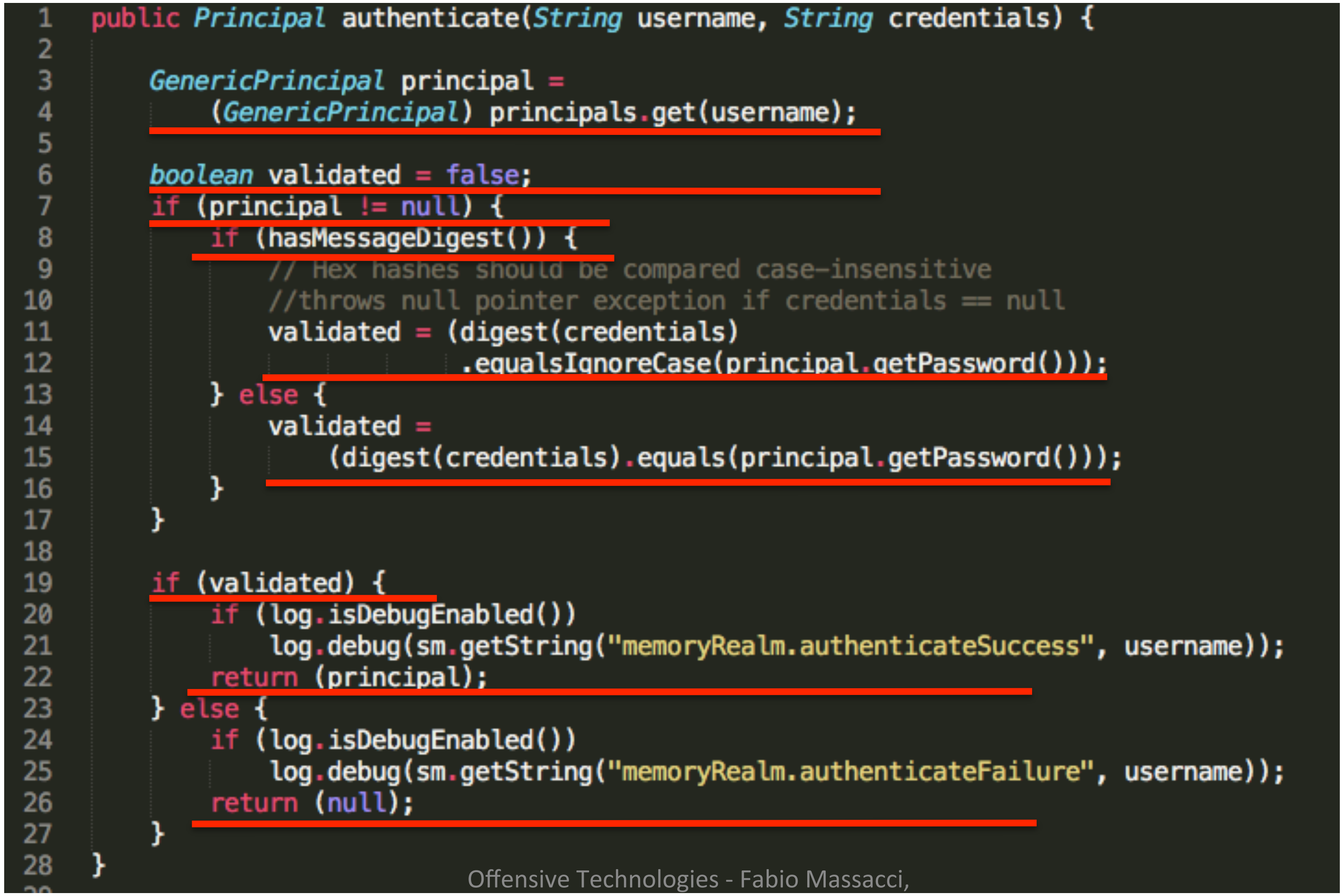
TASK 2 (CVE-2009-0580)

- **Information disclosure (user enumeration) in Apache Tomcat**
- Due to **insufficient error checking** in some **authentication** classes, Tomcat allows for the enumeration (brute force testing) of usernames by supplying illegally URL encoded passwords. The attack is possible if form based authenticaton (j_security_check) with one of the follow ingauthentication realms is used:
  - **MemoryRealm**
  - DataSourceRealm
  - JDBCRealm

UNIVERSITY OF TRENTO - Italy

eit Digital MASTER SCHOOL

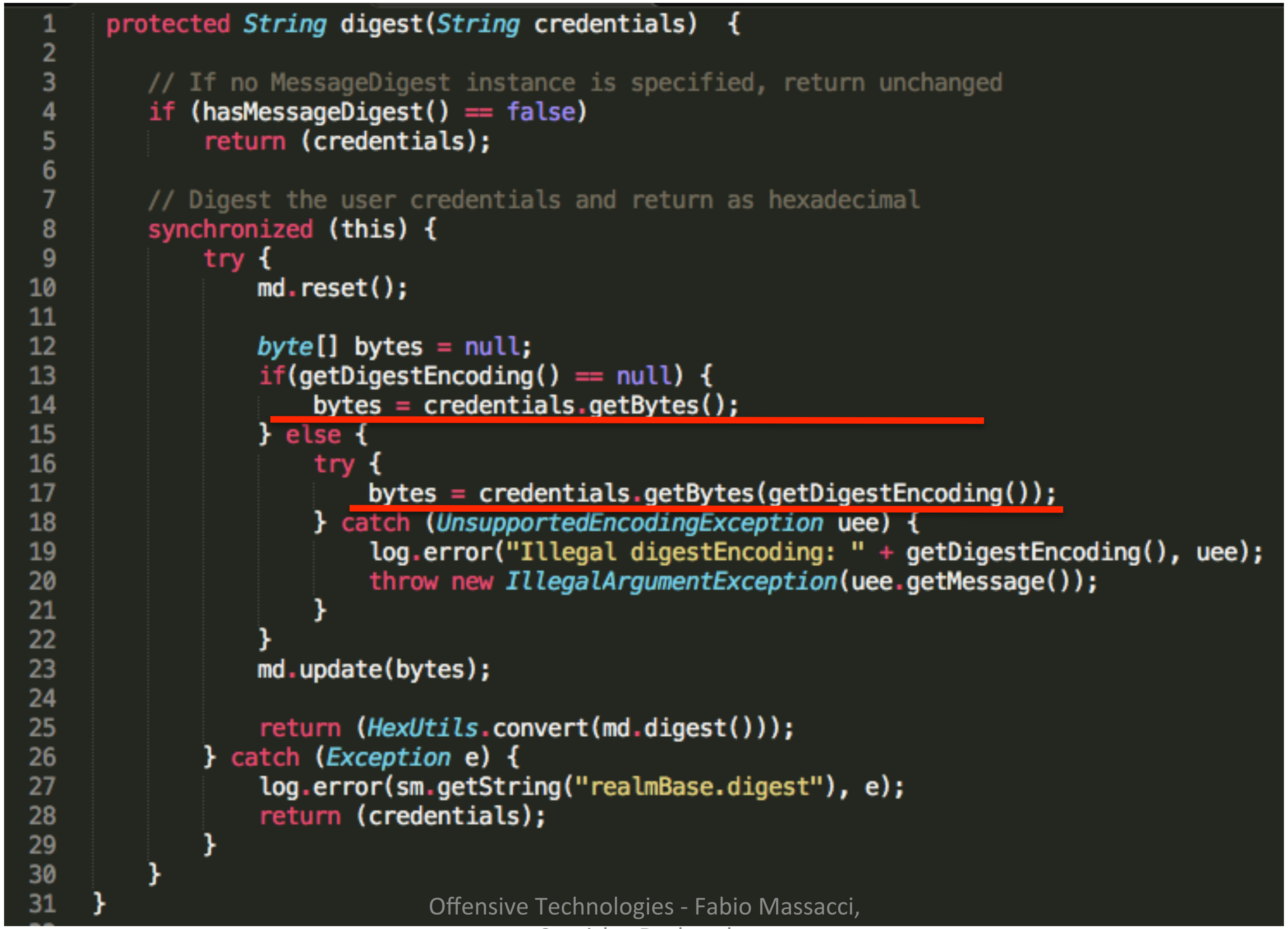
TASK 2 (CVE-2009-0580)



```
public Principal authenticate(String username, String credentials) {

    GenericPrincipal principal =
        (GenericPrincipal) principals.get(username);

    boolean validated = false;
    if (principal != null) {
        if (hasMessageDigest()) {
            // Hex hashes should be compared case-insensitive
            //throws null pointer exception if credentials == null
            validated = (digest(credentials)
                         .equalsIgnoreCase(principal.getPassword()));
        } else {
            validated =
                (digest(credentials).equals(principal.getPassword()));
        }
    }

    if (validated) {
        if (log.isDebugEnabled())
            log.debug(sm.getString("memoryRealm.authenticateSuccess", username));
        return (principal);
    } else {
        if (log.isDebugEnabled())
            log.debug(sm.getString("memoryRealm.authenticateFailure", username));
        return (null);
    }
}
```

UNIVERSITY OF TRENTO - Italy

TASK 2 (CVE-2009-0580)



```
protected String digest(String credentials)  {

    // If no MessageDigest instance is specified, return unchanged
    if (hasMessageDigest() == false)
        return (credentials);

    // Digest the user credentials and return as hexadecimal
    synchronized (this) {
        try {
            md.reset();

            byte[] bytes = null;
            if(getDigestEncoding() == null) {
                bytes = credentials.getBytes();
            } else {
                try {
                    bytes = credentials.getBytes(getDigestEncoding());
                } catch (UnsupportedEncodingException uee) {
                    log.error("Illegal digestEncoding: " + getDigestEncoding(), uee);
                    throw new IllegalArgumentException(uee.getMessage());
                }
            }
            md.update(bytes);

            return (HexUtils.convert(md.digest()));
        } catch (Exception e) {
            log.error(sm.getString("realmBase.digest"), e);
            return (credentials);
        }
    }
}
```

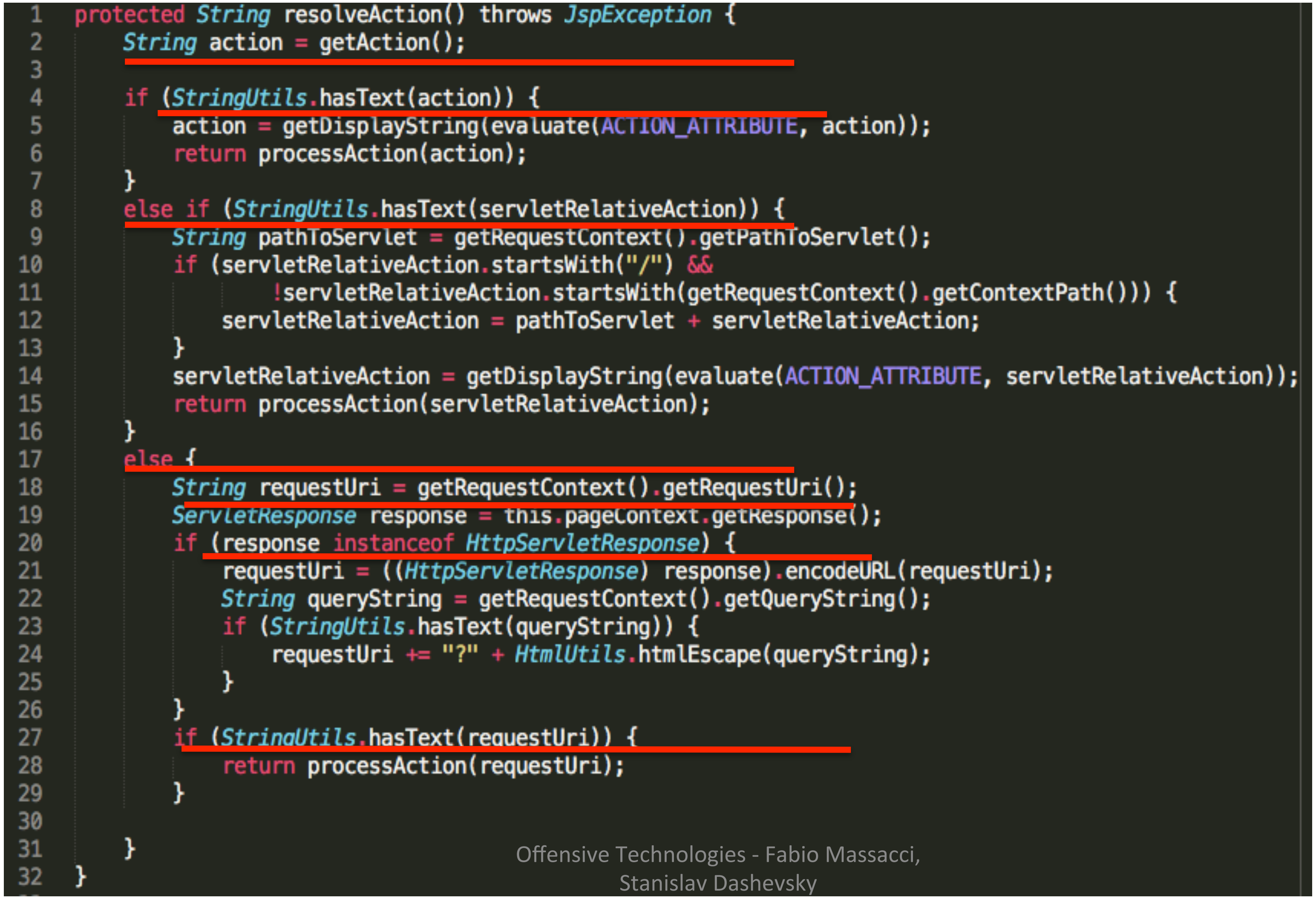
TASK 3 (CVE-2014-1904)

- **XSS in Spring Framework**
- Cross-site scripting (XSS) vulnerability in web/servlet/tags/form/FormTag.java allows remote attackers to **inject** arbitrary web script or HTML via the **requested URI** in a **default action.**

UNIVERSITY OF TRENTO - Italy

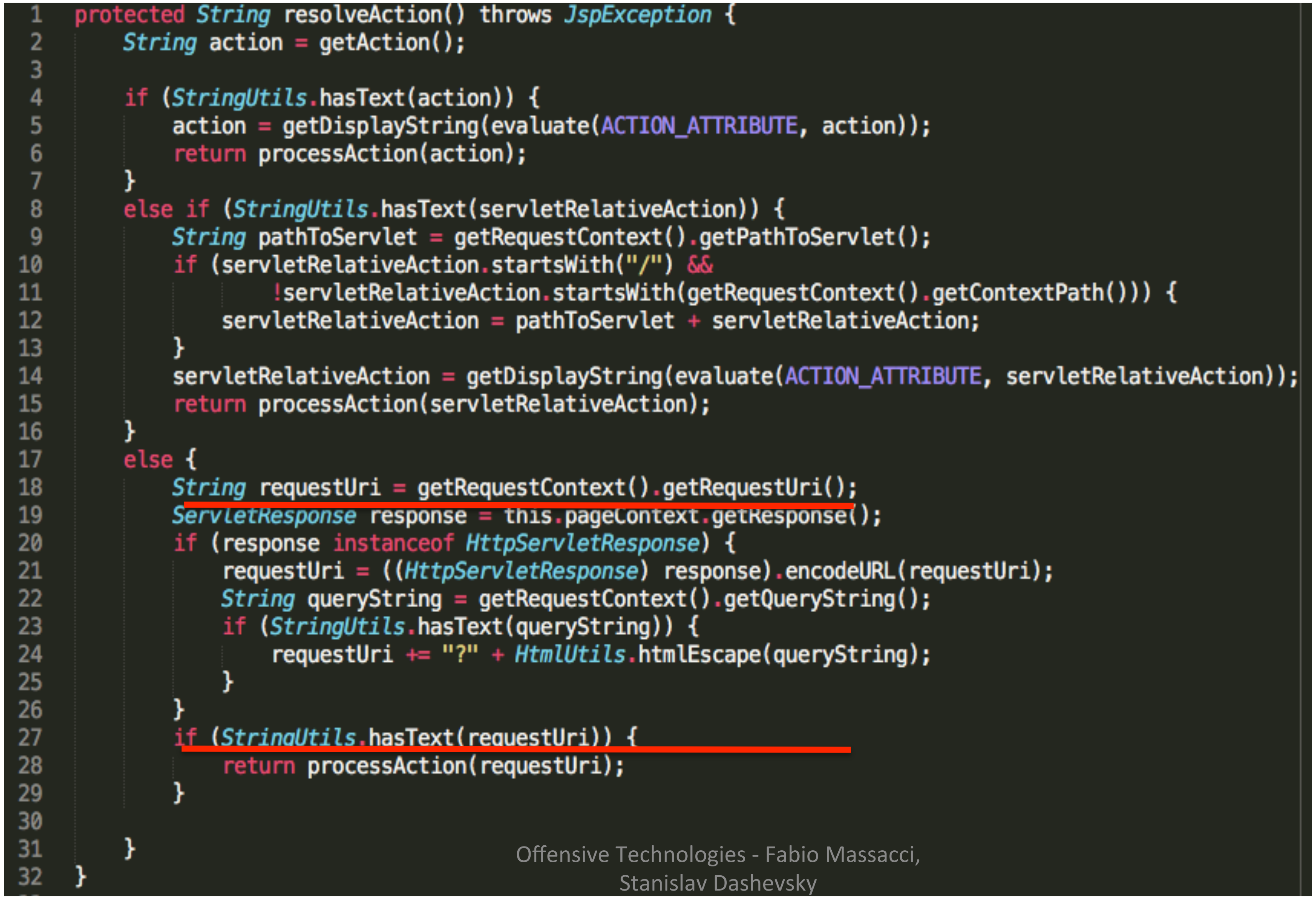
eit Digital MASTER SCHOOL

TASK 3 (CVE-2014-1904)



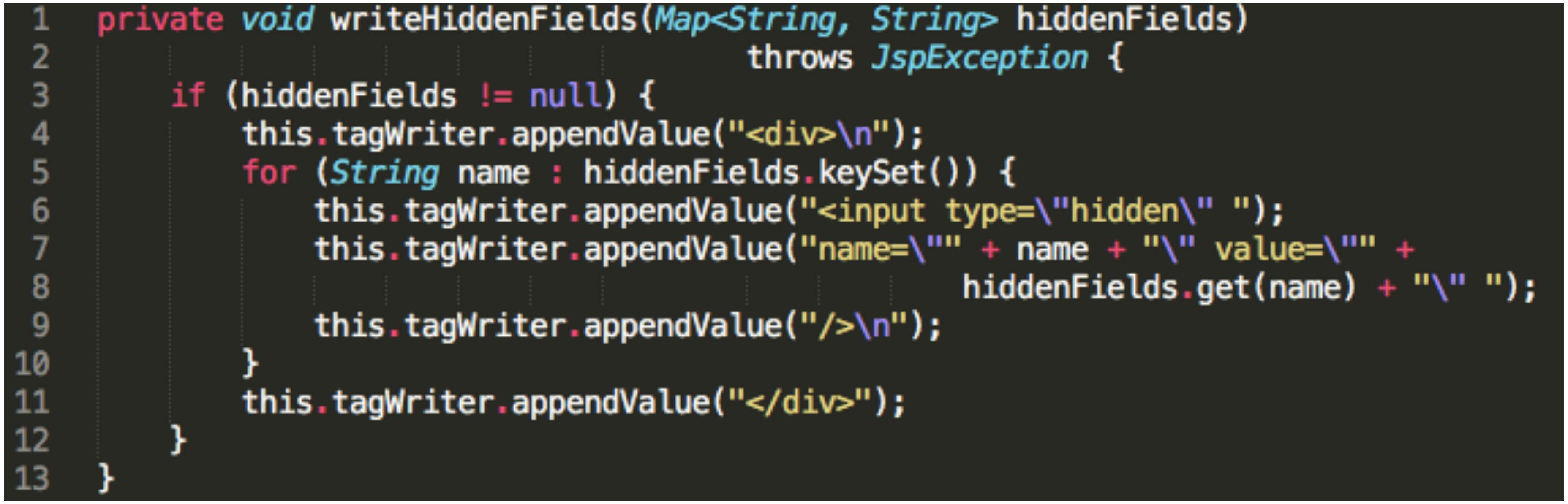
```java
protected String resolveAction() throws JspException {
    String action = getAction();

    if (StringUtils.hasText(action)) {
        action = getDisplayString(evaluate(ACTION_ATTRIBUTE, action));
        return processAction(action);
    }
    else if (StringUtils.hasText(servletRelativeAction)) {
        String pathToServlet = getRequestContext().getPathToServlet();
        if (servletRelativeAction.startsWith("/") &&
                !servletRelativeAction.startsWith(getRequestContext().getContextPath())) {
            servletRelativeAction = pathToServlet + servletRelativeAction;
        }
        servletRelativeAction = getDisplayString(evaluate(ACTION_ATTRIBUTE, servletRelativeAction));
        return processAction(servletRelativeAction);
    }
    else {
        String requestUri = getRequestContext().getRequestUri();
        ServletResponse response = this.pageContext.getResponse();
        if (response instanceof HttpServletResponse) {
            requestUri = ((HttpServletResponse) response).encodeURL(requestUri);
            String queryString = getRequestContext().getQueryString();
            if (StringUtils.hasText(queryString)) {
                requestUri += "?" + HtmlUtils.htmlEscape(queryString);
            }
        }
        if (StringUtils.hasText(requestUri)) {
            return processAction(requestUri);
        }

    }
}
```

TASK 3 (CVE-2014-1904)



```java
protected String resolveAction() throws JspException {
    String action = getAction();

    if (StringUtils.hasText(action)) {
        action = getDisplayString(evaluate(ACTION_ATTRIBUTE, action));
        return processAction(action);
    }
    else if (StringUtils.hasText(servletRelativeAction)) {
        String pathToServlet = getRequestContext().getPathToServlet();
        if (servletRelativeAction.startsWith("/") &&
                !servletRelativeAction.startsWith(getRequestContext().getContextPath())) {
            servletRelativeAction = pathToServlet + servletRelativeAction;
        }
        servletRelativeAction = getDisplayString(evaluate(ACTION_ATTRIBUTE, servletRelativeAction));
        return processAction(servletRelativeAction);
    }
    else {
        String requestUri = getRequestContext().getRequestUri();
        ServletResponse response = this.pageContext.getResponse();
        if (response instanceof HttpServletResponse) {
            requestUri = ((HttpServletResponse) response).encodeURL(requestUri);
            String queryString = getRequestContext().getQueryString();
            if (StringUtils.hasText(queryString)) {
                requestUri += "?" + HtmlUtils.htmlEscape(queryString);
            }
        }
        if (StringUtils.hasText(requestUri)) {
            return processAction(requestUri);
        }

    }
}
```

TASK 3 (CVE-2014-1904)

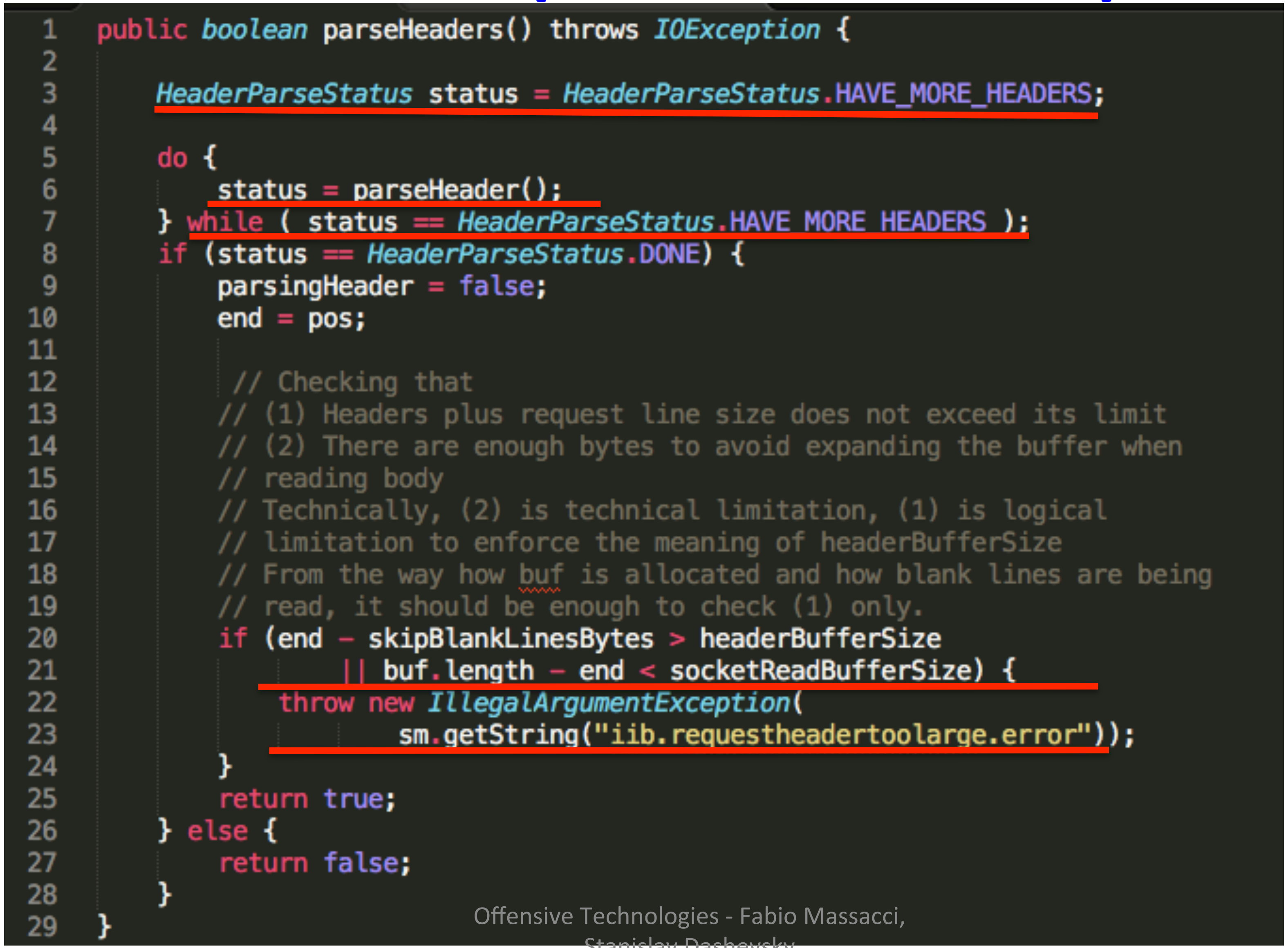


```java
private void writeHiddenFields(Map<String, String> hiddenFields)
                                            throws JspException {
    if (hiddenFields != null) {
        this.tagWriter.appendValue("<div>\n");
        for (String name : hiddenFields.keySet()) {
            this.tagWriter.appendValue("<input type=\"hidden\" ");
            this.tagWriter.appendValue("name=\"" + name + "\" value=\"" +
                                                    hiddenFields.get(name) + "\" ");
            this.tagWriter.appendValue("/>\n");
        }
        this.tagWriter.appendValue("</div>");
    }
}
```

TASK 4 (CVE-2012-2733)

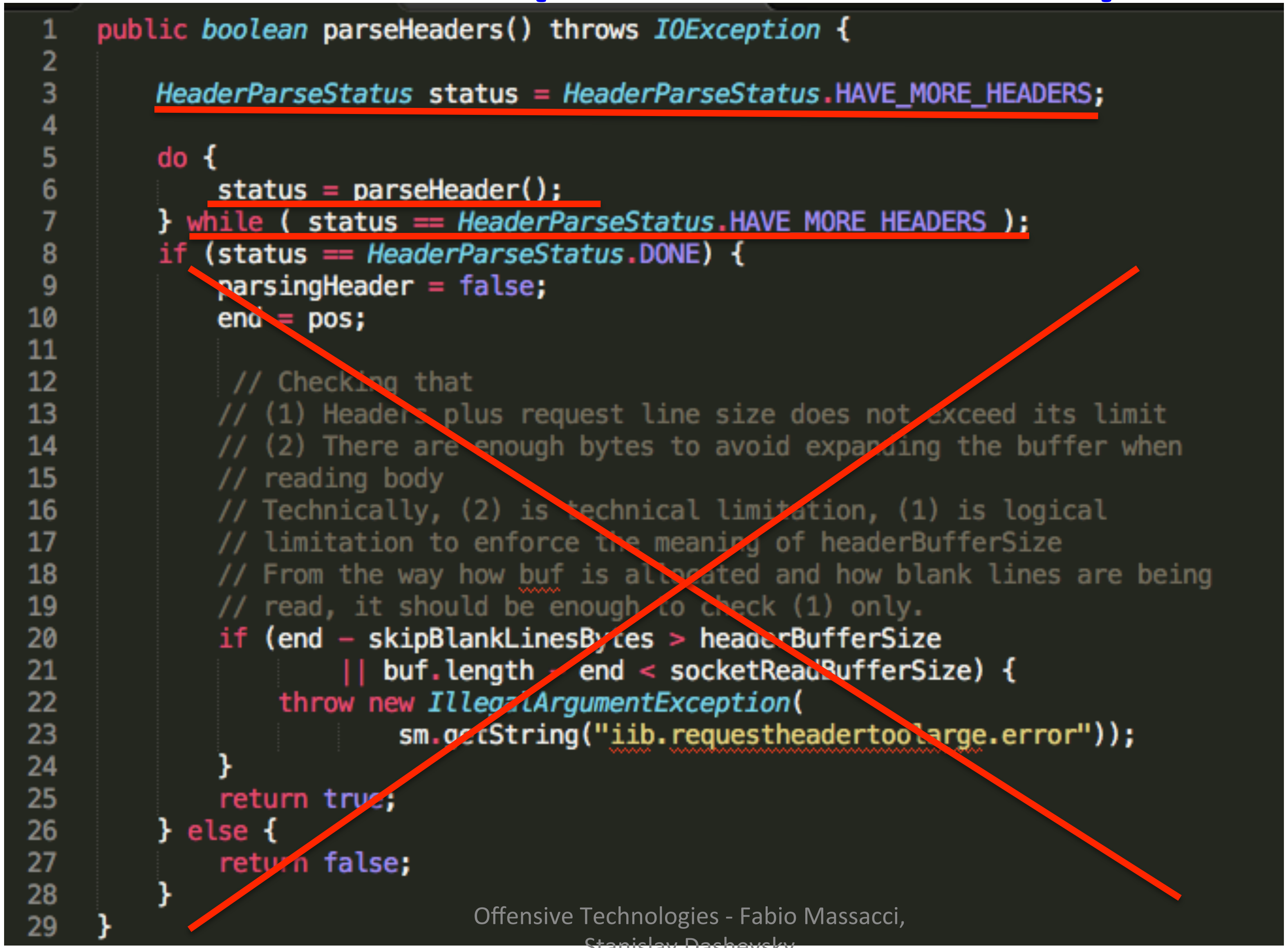
- **Denial of Service in Apache Tomcat**
- The **checks that limited the permitted size of request headers were implemented too late in the request parsing** process for the HTTP NIO connector. This enabled a malicious user to trigger an OutOfMemoryError by sending a single request with very large headers

TASK 4 (CVE-2012-2733)



```
public boolean parseHeaders() throws IOException {

    HeaderParseStatus status = HeaderParseStatus.HAVE_MORE_HEADERS;

    do {
        status = parseHeader();
    } while ( status == HeaderParseStatus.HAVE_MORE_HEADERS );
    if (status == HeaderParseStatus.DONE) {
        parsingHeader = false;
        end = pos;

         // Checking that
        // (1) Headers plus request line size does not exceed its limit
        // (2) There are enough bytes to avoid expanding the buffer when
        // reading body
        // Technically, (2) is technical limitation, (1) is logical
        // limitation to enforce the meaning of headerBufferSize
        // From the way how buf is allocated and how blank lines are being
        // read, it should be enough to check (1) only.
        if (end - skipBlankLinesBytes > headerBufferSize
                || buf.length - end < socketReadBufferSize) {
            throw new IllegalArgumentException(
                    sm.getString("iib.requestheadertoolarge.error"));
        }
        return true;
    } else {
        return false;
    }
}
```

TASK 4 (CVE-2012-2733)



```java
public boolean parseHeaders() throws IOException {

    HeaderParseStatus status = HeaderParseStatus.HAVE_MORE_HEADERS;

    do {
        status = parseHeader();
    } while ( status == HeaderParseStatus.HAVE_MORE_HEADERS );
    if (status == HeaderParseStatus.DONE) {
        parsingHeader = false;
        end = pos;

         // Checking that
        // (1) Headers plus request line size does not exceed its limit
        // (2) There are enough bytes to avoid expanding the buffer when
        // reading body
        // Technically, (2) is technical limitation, (1) is logical
        // limitation to enforce the meaning of headerBufferSize
        // From the way how buf is allocated and how blank lines are being
        // read, it should be enough to check (1) only.
        if (end - skipBlankLinesBytes > headerBufferSize
                || buf.length - end < socketReadBufferSize) {
            throw new IllegalArgumentException(
                    sm.getString("iib.requestheadertoolarge.error"));
        }
        return true;
    } else {
        return false;
    }
}
```