```
import java.util.*;
public class BadGenericThrow {
 public void makeFile() throws Exception {
  //create a Serializable List
  List<String> quarks = new ArrayList<String>();
  quarks.add("hello word");
  FileOutputStream file = null;
  ObjectOutputStream output = null;
  try{
   file = new FileOutputStream("quarks.ser");
   output = new ObjectOutputStream(file);
output.writeObject(quarks);
  }
  finally{
   if (output != null) {
    output.close();
   }
  }
 }
}
```

Forma corretta

```
import java.io.*;
import java.util.*;

public class BadGenericThrow {
 public void makeFile() throws  IOException, FileNotFoundException{
  //create a Serializable List
  List<String> quarks = new     ArrayList<String>();
  quarks.add("hello word");
  FileOutputStream file = null;
  ObjectOutputStream output = null;
  try{
   file = new FileOutputStream("quarks.ser");
   output = new ObjectOutputStream(file);
output.writeObject(quarks);
  }
  finally{
   if (output != null) {
    output.close();
   }
  }
 }
}
```

7.2.10.13Java Servlet

I dati dei moduli (form) html dovrebbero viaggiare preferibilmente attraverso richieste di tipo http POST, per cui il metodo doGet dovrebbe solo contenere la gestione dell'errore sollevato se viene invocato il metodo GET. La ragione per preferire il metodo POST sta nel fatto che in questo caso i parametri non viaggiano sull'url e non vengono pertanto memorizzati nei file di log o nella cache del browser. Chiaramente POST è ugualmente manipolabile, ma con maggior difficoltà.
Per implementare in maniera flessibile la sicurezza delle servlet si può agire attraverso la configurazione web.xml, oppure è possibile utilizzarel'annotazione @ServletSecurity con le annotazioni ausiliarie @HttpMethodConstraint e @HttpConstraint.

Nell'esempio seguente la servlet viene resa sicura attraverso la configurazione del file web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" [...] version="3.0">
  <display-name>Esempio Servlet Sicurezza</display-name>

  <servlet>
    <servlet-name>servletSicurezza</servlet-name>
    <servlet-class>com.package.servlet.servletSicurezza</servlet-class>
  </servlet>
```

```
  <servlet-mapping>
    <servlet-name>servletSicurezza</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>secure</web-resource-name>
      <url-pattern>/</url-pattern>
      <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>dipendente</role-name>
    </auth-constraint>
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>default</realm-name>
  </login-config>

</web-app>
```

Si impone l'autenticazione base, per il metodo GET. Solo gli utenti appartenenti al ruolo "dipendente" possono accedere.
La stessa cosa può essere ottenuta attraverso l'annotazione @ServletSecurity.
Esempi:
Il seguente codice impone l'uso della crittografia in tutte le connessioni GET

```
@ServletSecurity(@HttpConstraint(transportGuarantee =
TransportGuarantee.CONFIDENTIAL))
```

La dichiarazione seguente, invece, nega le connessioni con il metodo GET. POST invece è ammesso:

```
@ServletSecurity(
    httpMethodConstraints = @HttpMethodConstraint(value = "GET",
                        emptyRoleSemantic = EmptyRoleSemantic.DENY)
)
```

Con la seguente affermazione si permette l'acceso solo a utentti del ruolo "admin":

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "admin"))
```

I valori presenti come parametri nella request devono essere validati prima di poter essere utilizzati dall'applicazione. URL, Cookies, Form Fields, Hidden Fields, Headers, etc. ottenuti dai metodi getParameter(), getCookie(), o getHeader() degli oggetti HttpServletReques), prima di essere utilizzati, devono essere rigorosamente validate server-side.
Ciascun parametro in input deve specificare :

- il tipo di dato (string, integer, real, etc.);
- il set di caratteri consentito;
- la lunghezza minima e massima;
- la possibilità di accettare il valore NULL;
- la possibilità che il parametro sia richiesto o meno;
- la possibilità che siano permessi i duplicati;
- intervallo numerico;
- i valori ammessi (numerazione) ;
- i pattern (espressioni regolari) ;

Per ciascun parametro occorre inoltre filtrare qualsiasi carattere speciale e sostituirlo con il corrispondente carattere HTML. Queste routine di controllo devono essere sempre eseguite all'interno dei metodi doGet() e doPost() di tutte le Servlet che compongono la Web Application.
La tabella seguente mostra un'esemplificazione dei caratteri speciali che devono essere ricercati e le corrispondenti sostituzioni HTML che devono essere effettuate.

Carattere speciale da ricercare:	Carattere HTML sostitutivo:
<	&lt;
>	&gt;
(	&40;
)	&41;
#	&35;
&	&38;

Un metodo spesso adottato per effettuare queste operazioni di filtro e controllo consiste nell'adozione di un Web Application Firewall (WAF). Questi reverse proxies ricevono il traffico dai client, e dopo opportune modifiche, lo passano alla parte di back-end, sul server.

Anche per le servlet esiste la possibilità di una SQL injection. Un Web Application Firewall è una difesa contro le injection in generale, ma se una servlet accede ad un database, è opportuno che i relativi statements SQL non siano mai costruiti utilizzando concatenazioni di stringhe soprattutto se tali informazioni sono inserite dagli utenti. Invece della concatenazione dinamica di stringhe, occorre utilizzare l'interfaccia PreparedStatement che, tramite il driver JDBC, effettua la canonicalizzazione dei parametri in modo automatico. Di seguito un esempio di utilizzo dell'interfaccia PreparedStatement.

Esempio:

```
. . .
String selectStatement = "SELECT * FROM User WHERE userId = ? ";
PreparedStatement prepStmt = con.prepareStatement(selectStatement);
prepStmt.setString(1, userId);
ResultSet rs = prepStmt.executeQuery();
. . .
```

Per quanto riguarda il controllo delle sessioni, bisogna evitare di creare token di sessione ad-hoc ed utilizzare preferibilmente quelli messi a disposizione del web container (o web application server) in uso, gestendo le sessioni utente tramite l'apposita interfaccia javax.servlet.http.HttpSession. In qualsiasi caso i token di sessione dovrebbero sempre rispettare le seguenti regole:

- non devono mai essere inclusi nelle URL;
- devono essere costituiti da lunghe e complicate catene di numeri randomici che non possano essere facilmente indovinati;
- dovrebbero cambiare di frequente durante una sessione;
- dovrebbero cambiare quando si passa ad utilizzare protocolli come SSL;
- non devono mai essere utilizzati token scelti da un utente.

Se per memorizzare le sessioni si utilizzano i cookies, si devono sempre rispettare le seguenti regole minime:

- i cookies non devono essere mai utilizzati per memorizzare dati personali o informazioni sensibili (es: login, password, fede religiosa, malattie, etc.);
- prima di trasferire informazioni personali o sensibili verso un utente è necessario richiedere sempre la sua autenticazione e autorizzazione tramite inserimento di login e password: non basarsi mai sulla presenza o meno di un cookie precedentemente memorizzato;
- configurare i session cookies in modo tale che scadano quando l'utente esce dal browser;
- assicurarsi che tutte le informazioni contenute nei cookies siano accuratamente verificate e filtrate prima di essere utilizzate e/o inserite nei documenti HTML.

Altra norma che agevola la sicurezza consiste nel limitare la dimensione delle risposte http. Ove possible limitare sempre la lunghezza delle risposte HTTP al minimo necessario, troncando quelle che hanno una dimensione eccedente.

C'è anche una buona prassi che riguarda l'HTTP Referer. Ove possible, verificare sempre il campo Referer dell'header HTTP (es. metodo getHeader(java.lang.String name) dell'interfaccia