Linee guida sviluppo sicuro

Codifica documento: BLU-MAN-006-R01

Data emissione: 08/09/2026

INDICE

1 INTRODUZIONE AL DOCUMENTO 3
1.1 Scopo 3
1.2 Struttura del Documento 3
2 SECURE SOFTWARE DEVELOPMENT LIFE CYCLE (SSDLC) 4
3 DEV SEC OPS 6
4 DESIGN SICURO 7
4.1 ESEMPI 8
4.2 CONTROMISURE 10
5 ACCESS CONTROL 11
5.1 ESEMPI 11
5.1.1 Insecure Direct Object Reference (IDOR) 11
5.1.2 Cross-Site Request Forgery (CSRF) 12
5.1.3 API Access Control 13
5.1.4 SameSite Cookies, Same-Origin Policy e Cross-Origin Resource Sharing (CORS) 14
5.1.5 Content Security Policy (CSP) 16
5.2 CONTROMISURE 19
6 PRIVILEGE MANAGEMENT 19
6.1 ESEMPI 19
6.1.1 Privilegi di Default 19
6.1.2 Privilege Escalation Orizzontale e Privilege Escalation Verticale 20
6.2 CONTROMISURE 20
7 INPUT VALIDATION & SANITIZATION 20
7.1 ESEMPI 21
7.1.1 Command Injection 21
7.1.2 SQLi (SQL Injection) 21
7.1.3 Cross-Site Scripting (XSS) 22
7.1.4 Buffer Overflow 22
7.2 CONTROMISURE 22
8 GESTIONE DELLA CRITTOGRAFIA 23
8.1 ESEMPI 23
8.1.1 Gestione delle chiavi 23
8.1.2 Crittografia obsoleta 24
8.2 CONTROMISURE 25
9 CONFIGURAZIONI INSICURE 25
9.1 ESEMPI 25
9.2 CONTROMISURE 26
10 UTILIZZO DI CODICE DI TERZE PARTI 27
10.1 ESEMPI 27
10.2 CONTROMISURE 27
10.3 LOGGING E MONITORAGGIO 28
10.4 ESEMPI 28
10.5 CONTROMISURE 28
11 ELENCO ALLEGATI 28

1 INTRODUZIONE AL DOCUMENTO

1.1 Scopo

Bludigit, nel rispetto dei contratti di servizio stipulati, delle normative vigenti e secondo quanto stabilito da Italgas Holding, ha predisposto la presente normativa con l'obiettivo di definire le modalità e le linee guida da seguire nella gestione sicura e conforme dei processi IT e per il corretto utilizzo degli strumenti/apparati IT. Il documento si applica a Bludigit ed a tutte le società del Gruppo in funzione dei Contratti di Servizio infragruppo definiti.

Il documento illustra gli aspetti di sicurezza informatica nello sviluppo del codice applicativo, evidenziando l'importanza della protezione delle informazioni e della privacy degli utenti.

I controlli di sicurezza riportati nel presente documento si applicano ai sistemi informativi oggetto di gestione e, ove applicabile, agli ambienti di Operational Technology (OT), limitatamente ai sistemi SCADA centralizzati, in coerenza con i requisiti di sicurezza previsti dalla normativa NIS2 e nel rispetto delle specificità tecnologiche e dei vincoli operativi propri di tali contesti.
Per i sistemi OT sopra indicati, l'applicazione delle ***misure di hardening*** è definita secondo un approccio risk-based, proporzionato alla criticità dei processi industriali supportati e calibrato sui potenziali impatti in termini di sicurezza, continuità operativa e safety, in linea con il principio di proporzionalità sancito dalla normativa. Le configurazioni di riferimento sicure (security baseline) adottate per gli ambienti OT tengono conto dei vincoli tecnologici e di esercizio propri dei sistemi di controllo industriale, ivi inclusi i requisiti di disponibilità e di funzionamento in tempo reale dei processi produttivi.
Le presenti linee guida sono integrate, ove necessario, con le raccomandazioni dei fornitori e con gli standard e le best practice di settore applicabili agli ambienti industriali (es. famiglia IEC 62443), al fine di assicurare un livello di protezione adeguato e coerente con l'evoluzione del contesto normativo e tecnologico di riferimento.

Questo documento garantisce la conformità alle normative internazionali e locali, come ISO 27001, GDPR e AI Act, che regolamentano la sicurezza delle informazioni e l'uso etico delle tecnologie basate sull'intelligenza artificiale.

Le tematiche trattate riguardano le problematiche di sicurezza delle applicazioni moderne, con concetti applicabili a tutti i linguaggi. Ogni aspetto è accompagnato da esempi pratici e contromisure per prevenire vulnerabilità. L'obiettivo è aumentare la sicurezza dei sistemi aziendali e delle applicazioni, garantendo la protezione sin dalle fasi iniziali del ciclo di vita del software per mitigare rischi cyber e ridurre gli impatti su sistemi critici e dati sensibili. L'adozione di questi standard rafforza la fiducia di clienti e stakeholder. L'integrazione di principi di protezione nei processi IT supporta la crescita aziendale e protegge da sanzioni legali e danni reputazionali.

Bludigit promuove un approccio proattivo alla sicurezza informatica, allineato alle migliori pratiche di settore, ponendo la sicurezza al centro della strategia aziendale per garantire sostenibilità e competitività.

1.2 Struttura del Documento

Nel presente documento verranno trattate diverse tematiche, partendo da un *livello di astrazione* più alto andando man mano scendendo verso aspetti più pratici e tecnici, introducendo due processi tipici nel contesto dello sviluppo software orientato alla sicurezza, **SSDLC** e **DevSecOps**, evidenziando come le presenti linee guida vadano a collocarsi in ciascuna delle loro fasi.

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 pu 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 introdotto il concetto di **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 pu 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 **Input Validation & Sanitization**. 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 quindi spiegati attraverso esempi e scenari quali possono essere le diverse problematiche di sicurezza che possono scaturire da una mancata o scorretta validazione/sanitizzazione e come porvi rimedio.

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.

Inoltre, nello sviluppo software è una pratica molto comune l'**Utilizzo di Codice di Terze Parti**; anche questo tipo di codice è soggetto a problemi di sicurezza e pertanto è necessario assicurarsi di utilizzare librerie e codici di terze parti che siano verificate e che non presentino vulnerabilità note, effettuando periodiche review.

Infine, 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.

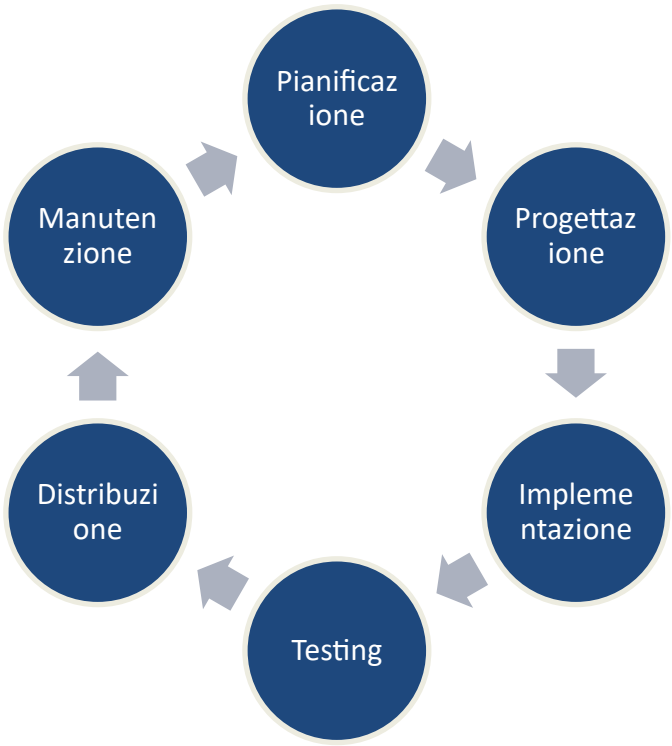
2 SECURE SOFTWARE DEVELOPMENT LIFE CYCLE (SSDLC)

Il **Secure Software Development Life Cycle** (SSDLC) è un approccio metodologico che integra la sicurezza nel processo di sviluppo del software fin dalle sue fasi iniziali. Questo modello si distingue dal tradizionale Software Development Life Cycle (SDLC) per l'attenzione costante alle questioni di sicurezza, che vengono affrontate in ogni fase, dalla pianificazione alla manutenzione e all'aggiornamento del software. L'obiettivo dell'SSDLC è quello di produrre software affidabile e sicuro, riducendo i rischi di vulnerabilità e attacchi informatici.

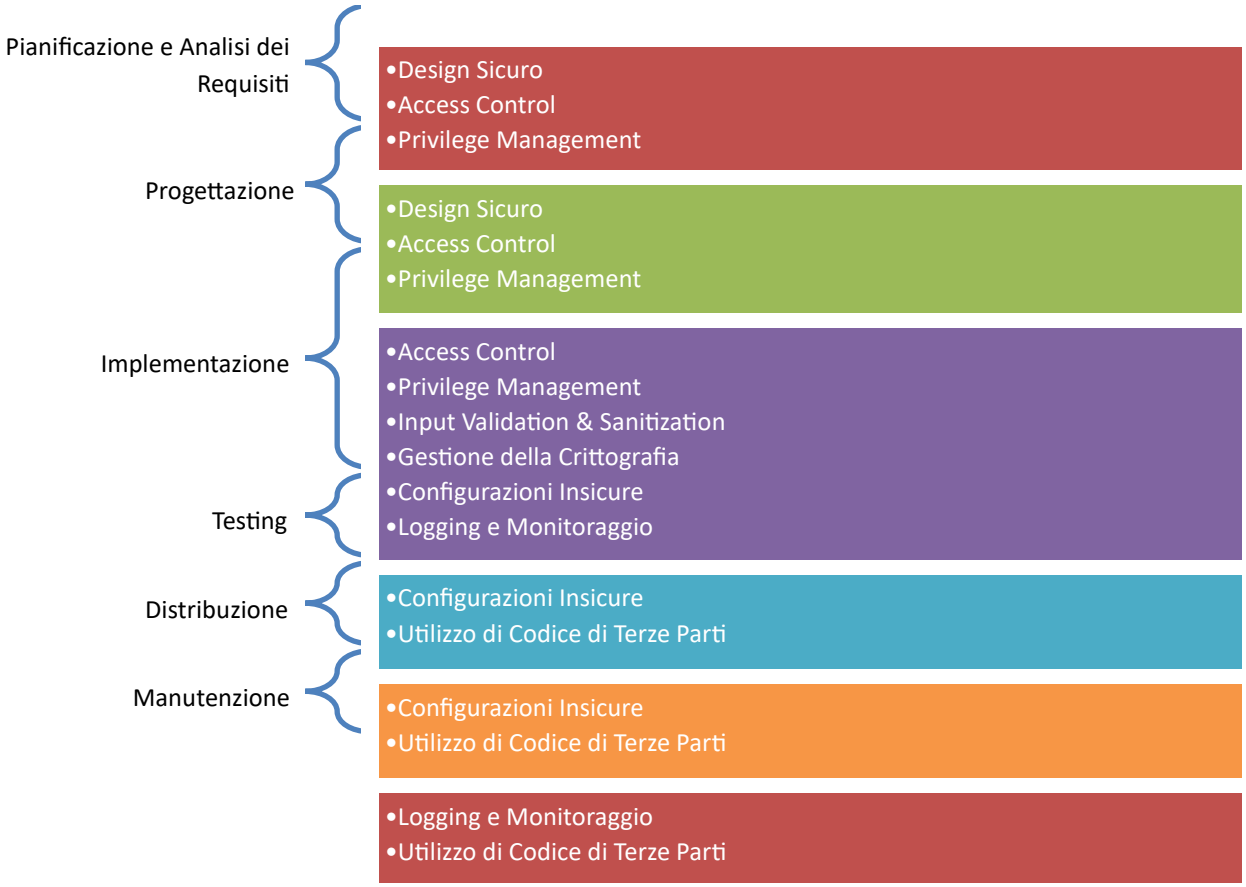
Le fasi del Secure Software Development Life Cycle (SSDLC) sono progettate per integrare la sicurezza nel processo di sviluppo del software fin dalle prime fasi. Ecco una panoramica delle fasi chiave:

- **Pianificazione e Analisi dei Requisiti**: Questa fase iniziale stabilisce le basi per l'intero progetto. Si tratta di definire cosa deve fare il software e come dovrebbe funzionare, identificando i requisiti di sicurezza specifici dell'applicazione.

- **Progettazione**: In questa fase si progetta la soluzione tecnica, tenendo conto dei requisiti di sicurezza, come l'autenticazione degli utenti, le restrizioni di accessibilità, le regole sulla privacy e l'auditability.
- **Implementazione**: Durante lo sviluppo del codice, si implementano le pratiche di codifica sicura per mitigare i rischi di sicurezza.
- **Testing**: I test di sicurezza vengono eseguiti per verificare che il software soddisfi tutti i requisiti di sicurezza e per identificare eventuali vulnerabilità. Questi possono consistere in analisi statica e/o dinamica del codice.
- **Distribuzione:** il software, dopo essere stato accuratamente testato e validato, viene distribuito nell'ambiente di produzione. Questa fase include l'implementazione di tutte le funzionalità di sicurezza pianificate, la verifica della conformità alle normative e agli standard di sicurezza, e la preparazione per il passaggio alla manutenzione e al monitoraggio continuo del software.
- **Manutenzione Continua**: Dopo il rilascio, il software continua a essere monitorato e aggiornato per affrontare nuove minacce alla sicurezza che possono emergere.



Le linee guida e le best practices presentate nel presente documento vanno a collocarsi come segue all'interno di tale processo:



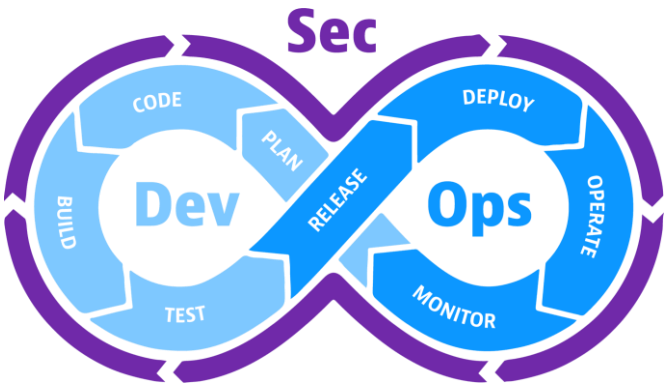
3 DEV SEC OPS

DevSecOps è un'evoluzione del DevOps che integra la sicurezza in ogni fase del ciclo di vita dello sviluppo software (SDLC). A differenza del DevOps, che si concentra su collaborazione ed efficienza tra sviluppo e operazioni, DevSecOps introduce la sicurezza fin dall'inizio del processo, noto come "shift left security". Questo approccio consente di rilevare e risolvere vulnerabilità precocemente, riducendo i costi e i rischi.

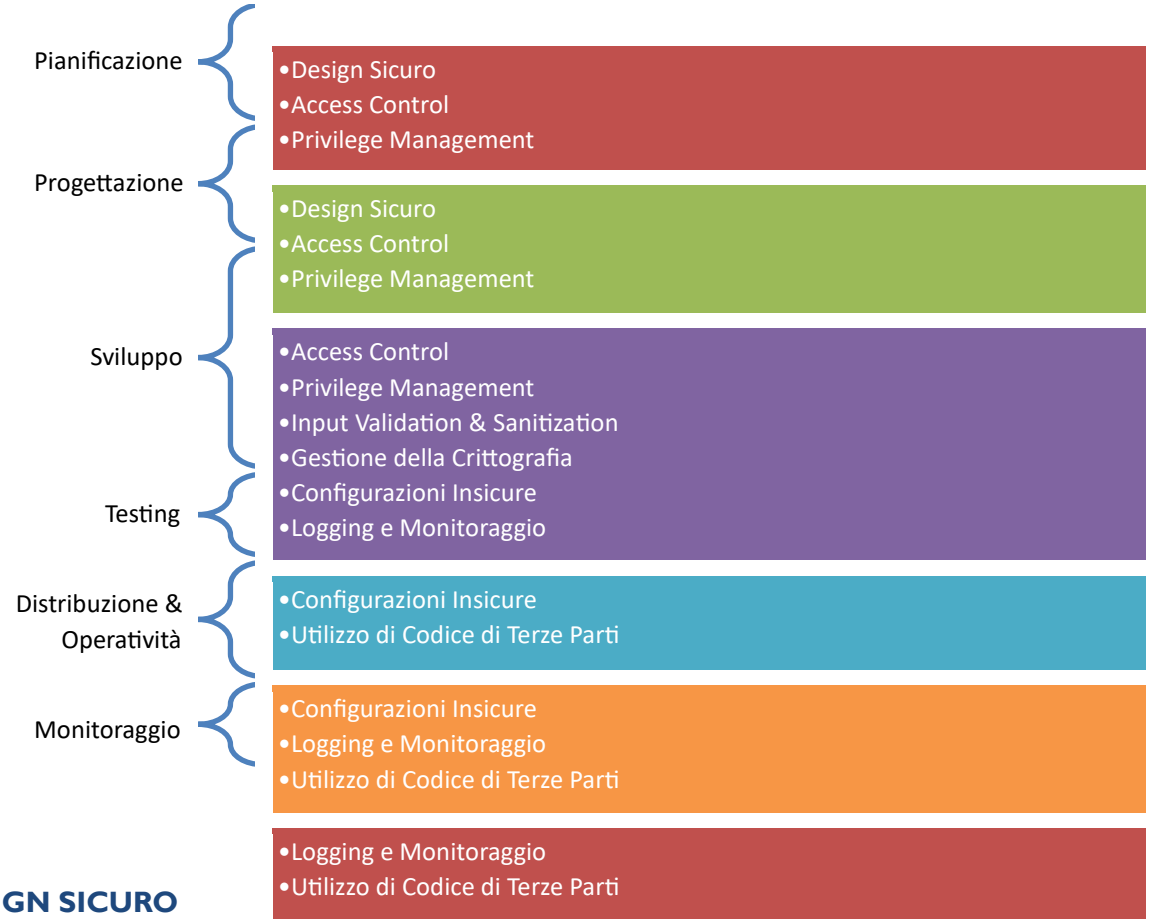
I principali benefici includono una sicurezza proattiva, con revisioni e test continui del codice per ridurre il rischio di violazioni e perdite di dati. Facilita anche la conformità normativa tramite test automatizzati e riduzione degli errori umani. Inoltre, promuove una cultura della sicurezza condivisa da tutto il team, non solo dagli esperti, permettendo lo sviluppo sicuro di nuove funzionalità lungo l'intero ciclo di sviluppo.

Le fasi di un processo DevSecOps includono:

- **Pianificazione**: Definizione dei requisiti di sicurezza e delle politiche da integrare nello sviluppo del software.
- **Progettazione**: Creazione di architetture software che includano controlli di sicurezza e valutazione dei rischi.
- **Sviluppo**: Scrittura del codice con l'integrazione di pratiche di sicurezza, come la revisione del codice e l'analisi statica e dinamica.
- **Testing**: Esecuzione di test automatizzati per identificare e correggere vulnerabilità di sicurezza prima della distribuzione.
- **Integrazione**: Automatizzazione dell'integrazione del codice e dei controlli di sicurezza per garantire la continuità e la qualità del software.
- **Distribuzione**: Rilascio del software in ambienti di produzione con processi automatizzati che includono la verifica della sicurezza.
- **Operatività**: Monitoraggio continuo della sicurezza del software in produzione e risposta rapida a eventuali incidenti.
- **Monitoraggio**: Sorveglianza costante delle prestazioni e della sicurezza del sistema per identificare e mitigare nuove vulnerabilità.



Le linee guida e le best practices presentate nel presente documento vanno a collocarsi come segue all'interno di tale processo:



4 DESIGN SICURO

Durante la fase di design di una nuova applicazione, è fondamentale adottare le best practices di sviluppo sicuro per evitare vulnerabilità. La mancata integrazione dei controlli di sicurezza sin dall'inizio pu introdurre rischi come accessi non autorizzati, corruzione dei dati e comportamenti anomali.

Aspetti chiave da considerare nel design applicativo includono:

- **Credenziali hardcoded**: Evitare di inserire password o chiavi direttamente nel codice, poiché espongono dati sensibili e complicano la gestione delle credenziali.
- **Gestione delle sessioni utente**: Assicurare connessioni sicure per prevenire accessi non autorizzati, hijacking delle sessioni e attacchi CSRF.

- **Gestione degli errori**: Implementare meccanismi sicuri per gestire errori senza rivelare dettagli interni dell'applicazione che potrebbero essere sfruttati da malintenzionati.
- **Accessi concorrenti in applicazioni multithread**: Prevenire condizioni di race e deadlock, che possono portare a incoerenza o corruzione dei dati.

Una delle attività che si suggerisce di eseguire in questa fase è l'analisi e la modellazione delle minacce (***Threat Modeling***). Quest'attività di sicurezza, spesso svolta nelle prime fasi di sviluppo, consente di individuare sin da subito eventuali falle nel design dell'applicativo.

4.1 ESEMPI

Di seguito si riportano degli esempi per i diversi aspetti affrontati al paragrafo precedente:

- **Credenziali "hardcoded"**: di seguito si riporta un esempio di credenziali hardcoded:

```
funzione accedi()     costante
username = "admin"
   costante password = "segretissima"

   leggi input_utente
   leggi input_password

   se input_utente = username e input_password =
password        stampa "Accesso consentito"     altrimenti
stampa "Accesso negato" fine funzione
```

- **Gestione delle sessioni utente**: di seguito si riporta un esempio di pseudocodice che illustra una cattiva gestione delle sessioni utente:

```
PROCESS AuthenticateUser
   INPUT username, password
   IF username == "admin" AND password == "password"
      SET loggedIn = true
   END IF
END PROCESS

IF loggedIn
   DISPLAY "Benvenuto, admin!"
ELSE
   REDIRECT_TO login_page END IF
```

  In questo pseudocodice l'utente viene autenticato tramite username e password. Tuttavia, non viene generato un token univoco per ogni utente. Se l'algoritmo di autenticazione fosse prevedibile (ad esempio, basato su credenziali semplici), un attaccante potrebbe determinarlo e impersonare altri utenti. Inoltre, Non viene gestito il trasporto sicuro del token. Se qualcuno intercettasse il token durante la trasmissione, potrebbe accedere all'account dell'utente.

- **Gestione degli errori**: di seguito si riporta un esempio di codice che va in eccezione non gestita. Nello specifico, l'array *arrayString* ha solo 10 elementi, ma stiamo cercando di accedere alla posizione

15, che non esiste. Di conseguenza, verrà lanciata un'eccezione di tipo *ArrayIndexOutOfBoundsException* che non viene gestita:

```
public class HelloException {     public
static void main(String[] args) {
        String[] arrayString = new String[10];
        System.out.println(arrayString[15]); // Genera un'eccezione di tipo ArrayIndexOutOfBoundsException
         }
}
```

Di seguito si propone un possibile gestione dell'eccezione per mezzo dell'utilizzo dei blocchi *try-catch* come suggerito al paragrafo successivo. Se l'eccezione viene lanciata (in questo caso, quando si cerca di accedere a un elemento inesistente nell'array), verrà catturata dal blocco catch, che gestirà l'eccezione stampando un messaggio appropriato. Così facendo si previene il crash dell'applicazione.

```
public class HelloException {     public
static void main(String[] args) {
String[] arrayString = new String[10];
try {
            System.out.println(arrayString[15]); // Genera un'eccezione di tipo ArrayIndexOutOfBoundsException
} catch (ArrayIndexOutOfBoundsException e) {
            // Gestisco l'eccezione
            System.out.println("Eccezione catturata");
        }
    }
}
```

- **Gestione degli accessi concorrenti in applicazione multithread**: i problemi spesso sorgono quando un thread esegue una "verifica quindi agisci" (ad esempio, "verifica" se il valore è X, quindi

"agisci" per fare qualcosa che dipende dal valore che è X) e un altro thread fa qualcosa al valore tra la "verifica" e l'"azione". Ad esempio:

```
if (x == 5):  # La "verifica"
y = x * 2  # L'"azione"

# Se un altro thread ha modificato x tra "if (x == 5)" e "y = x * 2" sopra, y potrebbe non essere uguale a 10.
```

4.2 CONTROMISURE

Di seguito si riportano alcune best practices da seguire afferenti ai punti elencati sopra:

- **Credenziali "hardcoded"**:
  - **Generazioni delle chiavi**: utilizzare algoritmi robusti per la generazione di nuove chiavi crittografiche;
  - **Gestione delle chiavi**: bisogna separare le credenziali e le chiavi dal codice, a tal fine si consiglia di utilizzare variabili di ambiente o servizi per la gestione delle chiavi;
  - **Archiviazione delle chiavi**: evitare di salvare le chiavi in file di testo non cifrati.
- **Gestione delle sessioni utente**:
  - **ID univoci**: ogni sessione dovrebbe avere un identificatore univoco (ad esempio, un token o un ID di sessione) per evitare collisioni e garantire l'integrità dei dati;
  - **Scadenza delle sessioni**: imposta un timeout per le sessioni inattive. Ad esempio, se un utente non interagisce con l'applicazione per un certo periodo di tempo, la sessione dovrebbe scadere automaticamente;
  - **Logout sicuro**: assicurarsi che il logout sia gestito correttamente. Quando un utente si disconnette, bisogna invalidare la sessione e cancellare eventuali dati sensibili associati;
  - **Gestione dei cookie**: utilizzare cookie sicuri e *HttpOnly* per memorizzare l'ID di sessione. I cookie sicuri sono trasmessi solo su connessioni *HTTPS*, mentre i cookie *HttpOnly* non possono essere letti da JavaScript, riducendo il rischio di attacchi XSS;
  - **Validazione dei dati di sessione**: verificare sempre l'ID di sessione fornito dall'utente per evitare sessioni forzate o manipolate;
  - **Prevenzione contro gli attacchi**: al fine di prevenire attacchi come l'hijacking delle sessioni, bisogna utilizzare token CSRF (Cross-Site Request Forgery) e proteggere i dati di sessione in transito utilizzando protocolli di comunicazione cifrati, ad esempio HTTPS.
- **Gestione degli errori**:
  - **Visualizzazione dei messaggi di errore**: bisogna fornire dei messaggi di errore descrittivi della problematica, ma semplici. Ovvero, bisogna evitare di mostrare dettagli tecnici che potrebbero essere sfruttati dagli attaccanti;
  - **Gestione delle eccezioni**: creare delle eccezioni personalizzate per situazioni specifiche. Si suggerisce di utilizzare dei blocchi *try-catch (Try-Except)* al fine di catturare gli errori e definirne una gestione sicura;
  - **Logging degli errori**: si suggerisce di registrare gli errori in un file di log dedicato e ad accesso protetto al fine di investigare nel dettaglio eventuali problematiche;
  - **Documentazione**: se si prevede che alcuni errori/eccezioni possano verificarsi, si consiglia di documentarli e di suggerire le procedure di risoluzione.
- **Gestione degli accessi concorrenti in applicazioni multithread**: ○ **Locks**: utilizzare meccanismi di lock, come *mutex* o *semafori*, al fine di garantire che un solo thread alla volta possa accedere a una risorsa condivisa;
  - **Atomicità**: Utilizzare operazioni atomiche per modificare variabili condivise, ad esempio incrementi o decrementi atomici.

5 ACCESS CONTROL

L'Access Control è il meccanismo che regola chi pu accedere a determinate risorse, garantendo che l'accesso sia appropriato e sicuro. L'assenza di un controllo efficace pu portare a violazioni di sicurezza come **data breach** e **violazioni della privacy**. Le policy di accesso devono bilanciare esigenze funzionali e di sicurezza, seguendo il principio dei **Privilegi Minimi**.

Esistono diversi modelli di Access Control:

- **Discretionary Access Control (DAC):** L'accesso è gestito a discrezione degli amministratori o proprietari delle risorse, usando Access Control Lists (ACL). È flessibile ma difficile da scalare e gestire a lungo termine.

- **Mandatory Access Control (MAC):** Un'autorità centralizzata regola l'accesso, senza possibilità di delega da parte dei proprietari delle risorse. È meno flessibile, ma garantisce coerenza e sicurezza, ed è usato in ambito governativo e militare.

- **Role-Based Access Control (RBAC):** L'accesso è determinato dal ruolo dell'utente all'interno dell'organizzazione. È più semplice da gestire, soprattutto in sistemi complessi.

- **Attribute-Based Access Control (ABAC):** L'accesso si basa su attributi dell'utente, della risorsa e del contesto. È altamente scalabile, ma meno efficace in assenza di attributi sufficienti.

5.1 ESEMPI

5.1.1 Insecure Direct Object Reference (IDOR)

L'**Insecure Direct Object Reference (IDOR)** è una vulnerabilità che si verifica quando i meccanismi di Access Control sono mal progettati, permettendo a utenti non autorizzati di accedere a risorse di sistema o dati di altri utenti. Questo accade tipicamente quando l'applicazione utilizza un input dell'utente, come un ID numerico, per accedere direttamente a un oggetto, ad esempio tramite un URL come: *`https://vulnerable.site/resources/resource?id=1336`*.
Un attaccante potrebbe sfruttare un attacco a forza bruta, inserendo diversi valori per l'ID, con l'intento di accedere a risorse non destinate a lui, approfittando della mancanza o dell'insufficienza dei meccanismi di controllo degli accessi.
Scenario di Esempio:

L'applicazione web "*vulnerable.site*" mette a disposizione una pagina in cui vengono riportate le informazioni personali dell'utente al seguente URL: *https://vulnerable.site/user/profile?id=**1336***.
Si desume che **1336** sia l'identificativo dell'utente attuale assegnato in maniera incrementale in fase di registrazione; un utente malintenzionato potrebbe effettuare una richiesta al seguente URL: *https://vulnerable.site/user/profile?id=**1337***.

Qualora non vi siano in atto meccanismi di controllo o questi risultino essere insufficienti, ad esempio il sistema di controllo verifica solo che l'utente sia un utente autenticato ma non che l'id richiesto sia quello di sua appartenenza, allora l'utente malintenzionato pu accedere ad informazioni riservate e potenzialmente sensibili di un altro utente dello stesso sistema. Un esempio in pseudo-codice di questa implementazione scorretta è il seguente:

```
funzione profilo() {
        if not session.user.isAuthenticated()
        then
            return NotAuthorizedCode
        end if
        user_id = request.id
```

```
        return get_profile_from_db(user_id)
}
```

La soluzione al problema di sicurezza precedente è quella di verificare che l'id richiesto appartenga effettivamente all'utente attualmente autenticato piuttosto che limitarsi a verificare che l'utente sia autenticato.

```
funzione profilo() {
        if not session.user.isAuthenticated() or request.id != session.user.id
        then
            return NotAuthorizedCode
        end if
        user_id = request.id
        return get_profile_from_db(user_id)
}
```

Una soluzione più corretta è quella di non esporre informazioni non necessarie in *URL* o body di richieste *POST*; in riferimento all'esempio, poiché l'informazione dell'identificativo dell'utente è già presente in sessione, sarebbe preferibile utilizzare direttamente il dato in sessione piuttosto che derivarlo dal parametro "id" nella URL (*https://vulnerable.site/user/profile*):

```
funzione profilo() {
        if not session.user.isAuthenticated()
        then
            return NotAuthorizedCode
        end if
    user_id = session.user.id      return get_profile_from_db(user_id)
}
```

Inoltre, piuttosto che utilizzare identificativi numerici, è preferibile utilizzare identificativi complessi, come ad esempio *GUID*, che rendono più complicati eventuali attacchi a forza bruta.

5.1.2 Cross-Site Request Forgery (CSRF)

Un attacco di **Cross-Site Request Forgery (CSRF)** sfrutta la fiducia che il browser ripone nell'utente, portando la vittima ad eseguire operazioni non volute a vantaggio dell'attaccante attraverso attacchi di phishing o social engineering, ad esempio per effettuare un cambio password/email, trasferire fondi all'attaccante o, nel caso in cui la vittima disponga di privilegi elevati, accedere eventualmente a funzionalità amministrative. Questa tipologia di attacco è attuabile quando mancano, o sono insufficienti, specifici meccanismi di controllo per le operazioni critiche e sensibili di un sistema.

Scenario di Esempio 1:

L'applicazione web "*vulnerable.site*" offre agli utenti la possibilità di effettuare il cambio email in autonomia effettuando la seguente richiesta HTTP:

```
POST /change-email HTTP/1.1
Content-Type= application/x-www-form-urlencoded
…

email=nuovaemail@email.com
```

Un attaccante potrebbe mirare ad impostare una mail sotto il suo controllo per un particolare utente, con lo scopo di, ad esempio, effettuare il reset della password ed entrare in possesso dell'account della vittima

(*Account Takeover*). Per poter raggiungere questo obiettivo, l'attaccante attraverso tecniche di phishing/social engineering fa visitare la vittima il seguente URL *https://evil.site*, il quale ospita il seguente codice:

```
<html>
<head>…</head>
<body>
        <form action="https://vulnerable.site/change-email" method="POST">
                <input type="hidden" name="email" value="evil-email@evil.com" />
        </form>
        <script>document.forms[0].submit()</script>
<body>
</html>
```

Il codice descritto invia automaticamente una richiesta POST al sito "vulnerable.site" per cambiare l'email, impostandone una sotto il controllo di un attaccante. Quando un utente visita la pagina, il browser invia la richiesta POST legittima con i cookie dell'applicazione, eseguendo il cambio email senza che l'utente lo sappia. La soluzione per prevenire questo attacco è l'integrazione di un **token CSRF**, una stringa randomica generata in modo sicuro, inclusa in un `<input type=hidden>` nel form. Se la richiesta non contiene un token valido, viene rigettata, impedendo l'attacco.

Scenario di Esempio 2:

In questo scenario, l'applicazione web "*vulnerable.site*" integra un *token CSRF* per la richiesta di cambio email:

```
POST /change-email HTTP/1.1
Content-Type= application/x-www-form-urlencoded
…

email=nuovaemail@email.com&csrf-token=WSVeBWnAJaGddAuZ9cVhWVIsF9WIY4rCT9Ab4YEAs64
```

Sebbene non sia possibile per un attaccante generare un token CSRF valido in autonomia, quando il token CSRF non è legato alla sessione dell'utente per il quale è stato generato, è sufficiente per l'attaccante farsi fornire dall'applicazione un token CSRF nella propria sessione ed utilizzarlo poi per condurre l'attacco utilizzando lo stesso token nella sessione della vittima:

```
<html>
<head>…</head>
<body>
        <form action="https://vulnreable.site/change-email" method="POST">
                <input type="hidden" name="email" value="evil-email@evil.com" />
                <input type="hidden" name="csrf-token" value=
"PuW1Tk1FViPqhXsyIvsGJamY5ZP2a/crHZTbqNNkxfQ" />
        </form>
        <script>document.forms[0].submit()</script>
<body> </html>
```

Nel caso descritto, un token CSRF (ad esempio "PuW1Tk1FViPqhXsyIvsGJamY5ZP2a/crHZTbqNNkxfQ") viene generato per l'attaccante. Quando un utente visita la pagina, il browser invia automaticamente la richiesta POST con i cookie legittimi, e l'applicazione "vulnerable.site" esegue il cambio email. La soluzione consiste nell'associare ogni token CSRF alla sessione dell'utente, rendendo valido il token solo per la sessione corrente. Meccanismi come SameSite Cookie o Cross-Origin Resource Sharing, se implementati correttamente, possono prevenire gli attacchi CSRF. Tuttavia, è fondamentale integrare diverse misure di sicurezza, seguendo il principio di Defense in Depth, in modo che la sicurezza del sistema resti solida anche se uno dei livelli fallisce.

5.1.3 API Access Control

Le **API (Application Programming Interfaces)** sono protocolli che permettono a software diversi di comunicare tra loro, facilitando l'accesso a servizi o dati in modo standardizzato. Sono fondamentali per

creare sistemi modulari e scalabili, permettendo di inviare e ricevere dati da varie fonti, indipendentemente dalle tecnologie sottostanti. Tuttavia, una progettazione insicura delle API pu compromettere l'integrità del sistema e mettere a rischio la riservatezza dei dati.

Scenario di Esempio I:

Il front-end di "vulnerable.site" richiama delle API per fornire servizi agli utenti, come l'endpoint `https://vulnerable.site/api/v1/users/my-profile` per ottenere le informazioni del profilo dell'utente autenticato. Tuttavia, in una progettazione insicura, l'endpoint `https://vulnerable.site/api/v1/users` permette di ottenere informazioni su tutti gli utenti, esponendo dati sensibili. Un attaccante potrebbe interrogare questo endpoint, anche se non utilizzato esplicitamente dal front-end. La soluzione consiste nell'implementare adeguate policy di controllo degli accessi, per garantire che ogni utente acceda solo alle informazioni necessarie durante l'interazione con il sistema.

Scenario di Esempio 2:

In uno scenario in cui una nuova versione delle API (v2) risolve un problema di sicurezza, è comune dover mantenere attive anche le versioni precedenti (v1) per motivi di retrocompatibilità, ad esempio con app mobile che non sono aggiornate uniformemente. Questo pu portare a una falsa supposizione che nessuno acceda più alla versione vulnerabile. Tuttavia, un attaccante potrebbe continuare a sfruttarla. La soluzione è limitare il ciclo di vita delle versioni deprecate al minimo necessario e applicare patch di sicurezza anche su queste versioni obsolete per mitigare i rischi.

Scenario di Esempio 3:

Nell'applicazione "vulnerable.site", esistono API amministrative accessibili tramite l'endpoint `https://vulnerable.site/api/v2/private`, che non sono esposte agli utenti senza privilegi. Si potrebbe pensare che non siano necessari controlli su queste API, ma un attaccante potrebbe usare tecniche di enumerazione e fuzzing per scoprire endpoint nascosti.
Questo approccio, basato sul principio di "**Security through Obscurity**", è rischioso poiché non garantisce una protezione adeguata. La soluzione consiste nell'implementare meccanismi di controllo e autorizzazione robusti su tutte le funzionalità del sistema, e un sistema di **throttling** per limitare le richieste da parte di un singolo client, prevenendo così anche attacchi *Denial of Service (DoS)* e ottimizzando le risorse del server.

5.1.4 SameSite Cookies, Same-Origin Policy e Cross-Origin Resource Sharing (CORS)

I cookie **SameSite** sono un meccanismo introdotto per istruire i browser su quando associare o meno i cookie memorizzati in esso a richieste HTTP generate a partire da un sito first-party verso un sito di terze parti, vale a dire richieste *cross-origin*. Questo meccanismo permette di prevenire e mitigare diversi attacchi come, ad esempio, di **Cross-Site Request Forgery**.

Tale meccanismo è implementato associando ad un cookie l'attributo **SameSite**, il quale pu assumere tre valori:

- **None**: il cookie verrà incluso in tutte le richieste cross-origin.
- **Lax**: il cookie verrà incluso solo in richieste HTTP cross-origin con metodo GET e derivanti da una interazione con l'utente, ad esempio, un click su un link.
- **Strict**: il cookie non viene mai incluso in richieste HTTP cross-origin.

Un esempio di utilizzo di tale meccanismo è il seguente:

```
Set-Cookie: session=LnppYMJsSafNatxtcYjAYylRwsbRNZxo; SameSite=Strict
```

Di default, i browser moderni, quando non esplicitamente settato l'attributo *SameSite*, trattano il cookie con policy *Lax*. L'utilizzo della policy *None* è scoraggiato ed è comunque vincolato all'utilizzo concomitante dell'attributo *Secure* che indica al browser di includere il cookie solo su canali di comunicazione sicuri (es:

HTTPs). La scelta tra *Lax* e *Strict* è determinata commisurando le esigenze funzionali e quelle di sicurezza; la prima offre una flessibilità maggiore, la seconda è più rigorosa.

La **Same-Origin Policy** è un meccanismo di sicurezza che regola come un documento o uno script presenti su una origine possono interagire con risorse di altre origini. Questo permette di prevenire che una certa origine venga sfruttate da attori malevoli per condurre attacchi verso altre origini. Due origini sono considerate la stessa origine (*same-origin*) quando protocollo, porta ed host sono gli stessi.

https://company.site/subdir1 VS https://company.site/subdir2 = Stessa origine
https://company.site:8080/subdir1 VS https://company.site/subdir1 = Origine diversa (porta differente)
http://company.site/subdir1 VS https://company.site/subdir1 = Origine diversa (protocollo differente)

L'implementazione della **Same-Origin Policy** prevede che:

- È permesso l'embedding cross-origin di pagine web tramite iframe, ma non l'accesso al documento contenuto in un iframe via JavaScript.
- È permesso l'embedding di CSS cross-origin.
- È permesso il submit di form con action cross-origin.
- È permesso l'embedding cross-origin di immagini, ma non l'accesso ai dati binari tramite JavaScript.
- È permesso l'embedding di elementi multimediali come audio e video.
- È permesso l'embedding di script cross-origin, ma è bloccato l'accesso ad API che effettuano richieste HTTP verso origini differenti.

Il **Cross-Origin Resource Sharing (CORS)** è stato introdotto per controllare gli accessi a risorse di domini di terze parti, rappresentando un rilassamento della Same-Origin Policy. Se mal configurato, pu portare a vulnerabilità come attacchi cross-domain. L'implementazione di CORS avviene tramite header nelle risposte HTTP, che istruiscono i browser su come gestire gli accessi.

L'header **Access-Control-Allow-Origin** specifica quale origine pu accedere alla risorsa. Pu essere:

- ***:** consente l'accesso da qualsiasi origine (solo per richieste senza credenziali).
- **<Origine>:** consente l'accesso da una sola origine specifica, ad esempio `https://company.site`.
- **null:** sconsigliato, poiché pu essere sfruttato in particolari scenari per generare richieste con `Origin` "null".

L'header **Access-Control-Allow-Credentials** indica se è consentito includere credenziali (cookie, header Authorization) nelle richieste cross-origin. Pu essere `true` o `false`.

Gli header **Access-Control-Allow-Methods** e **Access-Control-Allow-Headers** specificano i metodi HTTP e gli header necessari per l'accesso a risorse cross-domain, forniti dal server tramite una richiesta "preflight" con metodo OPTIONS.

Sebbene CORS renda la Same-Origin Policy più flessibile, una configurazione errata pu portare a problemi di sicurezza come l'esfiltrazione dei dati e accessi non autorizzati.

Scenario di Esempio 1:

L'applicazione di terze parti "*https://vulnerable.site*" implementa il meccanismo di CORS, specificando nelle sue risposte l'header *Access-Control-Allow-Origin*.

Tuttavia, per poter supportare l'interazione con *molteplici* origini esterne (Access-Control-Allow-Origin ammette una sola origine), tale header viene avvalorato dinamicamente riflettendo il valore dell'header ***Origin*** nella richiesta.

Questa implementazione abbatte difatti lo scopo del meccanismo di CORS stesso, aprendo diverse possibilità di attacco per utenti malintenzionati. Una corretta implementazione, invece, prevede di validare preferibilmente con l'ausilio di *whitelists* le origini nelle richieste, ammettendo solo origini fidate.

Scenario di Esempio 2:

L'applicazione "https://vulnerable.site" utilizza CORS, impostando gli header `Access-Control-Allow-Origin` e `Access-Control-Allow-Credentials` su `true`, con il valore dinamico di `Access-Control-Allow-Origin` che varia tra "http://trusted.origin" o "https://trusted.origin" a seconda del protocollo della richiesta. Un attaccante pu esfiltrare informazioni sensibili inducendo il browser a fare una richiesta cross-origin tramite HTTP, intercettando il traffico non cifrato e rubando dati come cookie o token di autenticazione. La soluzione consiste nell'evitare l'uso di HTTP non sicuri o, quando inevitabile, nel prevenire l'inclusione di credenziali in comunicazioni non sicure.

5.1.5 Content Security Policy (CSP)

La Content Security Policy (CSP) è una misura di sicurezza che protegge le applicazioni web da attacchi come il Cross-Site Scripting (XSS), che possono causare furto di dati, alterazione di siti web e diffusione di malware. Implementando la CSP, gli amministratori definiscono quali domini sono considerati affidabili per l'esecuzione di script e il caricamento di risorse, limitando i vettori per attacchi XSS.
La CSP promuove anche l'uso di HTTPS per una trasmissione sicura dei dati. La configurazione della CSP avviene tramite l'header HTTP `Content-Security-Policy` o tramite l'elemento `<meta>` in una pagina HTML. Le direttive CSP specificano quali risorse possono essere caricate e da quali domini e protocolli. Segue un elenco delle direttive più comuni:

- **default-src**: Specifica le origini predefinite per tutti i tipi di risorse, se non vengono specificate altre direttive.
- **script-src**: Definisce le origini valide da cui possono essere caricati gli script.
- **style-src**: Controlla le origini da cui possono essere caricati gli stili CSS.
- **img-src**: Stabilisce le origini da cui possono essere caricate le immagini.
- **connect-src**: Limita le origini a cui è possibile connettersi (ad esempio, tramite XMLHttpRequest, WebSocket o fetch()).
- **font-src**: Specifica le origini da cui possono essere scaricati i font.
- **object-src**: Controlla le origini da cui possono essere caricati gli elementi <object>, <embed> e <applet>.
- **media-src**: Definisce le origini da cui possono essere caricate risorse multimediali come audio e video.
- **frame-src**: Determina le origini da cui possono essere caricati i frame.
- **child-src**: Deprecata e sostituita da frame-src, era utilizzata per definire le origini da cui possono essere caricati i frame e i worker.
- **form-action**: Specifica le origini a cui possono essere inviate le form submission.
- **base-uri**: Restringe le URL che possono essere utilizzate nel tag <base>.
- **report-uri**: Indica agli user agent dove inviare i report di violazione della policy.

Segue un esempio di implementazione di una CSP:

```
Content-Security-Policy: default-src 'self'; script-src 'self' https://*.esempio.com; object-src 'none'; upgrade-insecure-requests; report-uri /csp-report-endpoint/`
```

La policy stabilisce che script e CSS devono provenire dallo stesso dominio del sito, ma gli script possono anche essere caricati da tutti i sottodomini di `esempio.com` usando una **wildcard (*)**. La direttiva `objectsrc` vieta oggetti come `<object>`, `<embed>` o `<applet>`. Le richieste HTTP vengono

automaticamente convertite in HTTPS per migliorare la sicurezza, e i tentativi di violazione della CSP sono segnalati a un endpoint specifico.

Tuttavia, la policy potrebbe non prevenire vulnerabilità, ad esempio se una CDN esterna come `cdn.esempio.com` carica uno script malevolo, che verrebbe considerato affidabile dalla CSP. Una soluzione parziale consiste nell'esplicitare i domini necessari, evitando l'uso di wildcard, specialmente quando i domini non sono sotto il proprio controllo.

Nello scenario in cui caricare script da *cdn.esempio.com* sia necessario per il corretto funzionamento del sito, è bene utilizzare un approccio differente. In questo caso ci sono due alternative di approccio:

- Nonce-based
- Hash-based

Approccio Nonce-Based

Un **nonce** è un numero casuale generato dal server per ogni richiesta HTTP, incluso nell'header CSP e associato ai tag di script o stile nella pagina. Il browser eseguirà solo gli script e gli stili con un attributo nonce corrispondente al valore nell'header CSP, impedendo l'esecuzione di script malevoli tramite XSS. Il nonce deve essere casuale e imprevedibile per garantire la sicurezza. Inoltre, la direttiva `**strict-dynamic**` della CSP permette agli script approvati di caricare altre risorse dinamicamente, aumentando la sicurezza e la flessibilità, particolarmente quando una CDN carica ulteriori risorse.

Segue un esempio di utilizzo del nonce:

```
// Nell'header di risposta HTTP
Content-Security-Policy: script-src 'strict-dynamic' 'nonce-abc123'

//Nella pagina HTML
<script src="cdn.esempio.com" nonce="abc123"></script>
```

Approccio Hash-Based

L'approccio basato su hash permette di definire una lista di hash crittografici corrispondenti agli script inline o esterni che si desidera autorizzare.
Ogni volta che uno script viene eseguito, il suo hash viene calcolato e confrontato con l'elenco degli hash definiti nella CSP.
Se l'hash corrisponde, lo script viene eseguito; in caso contrario, viene bloccato.

Questo metodo è particolarmente utile per le pagine servite staticamente o per quelle che necessitano di essere memorizzate nella cache.

Per generare un hash, si pu utilizzare un comando OpenSSL o strumenti simili, e l'hash risultante viene incluso nell'intestazione HTTP della CSP.
È importante notare che ogni modifica allo script richiederà la generazione di un nuovo hash.

```
// Nell'header di risposta HTTP
Content-Security-Policy: script-src 'sha256-Xyz123=='

//Nella pagina HTML
<script src="https://cdn.esempio.com/somescript.js"></script>
```

Nell'esempio precedente, l'hash dello script *somescript.js* utilizzando come algoritmo di hashing SHA-256 deve corrispondere a *Xyz123*== per poter essere eseguito.

Trusted Types

I Trusted Types sono una funzionalità di sicurezza del web che aiuta a prevenire gli attacchi di tipo Cross-Site Scripting (XSS) manipolando i tipi di dati che possono essere inseriti nel Document Object Model (DOM). Questi permettono agli sviluppatori di definire delle politiche di sicurezza che specificano quali tipi di dati possono essere considerati affidabili e quindi accettati dal DOM.

Questo si integra con la Content Security Policy (CSP) attraverso la direttiva *trusted-type*.

Segue un esempio di utilizzo dei trusted types:

```
// Definizione della politica Trusted Types
if (window.trustedTypes && trustedTypes.createPolicy) {
window.trustedTypes.createPolicy('myPolicy', {
createHTML: (string) => string
  });
}

// Utilizzo della politica Trusted Types nel codice JavaScript
const trustedHtml = trustedTypes.createPolicy('myPolicy', {
createHTML: (string) => {
    // Sanitizzazione della stringa
    const sanitizedString = DOMPurify.sanitize(string);
return sanitizedString;
  }
}).createHTML('<script>alert(1)</script>');

// Inserimento del Trusted HTML nel DOM
document.body.innerHTML = trustedHtml;
```

In questo esempio, viene definita una politica Trusted Types chiamata 'myPolicy', che permette la creazione di HTML sicuro utilizzando la libreria di sanitizzazione *DOM Purify*. Il codice HTML contenente uno script Javascript verrà quindi sanitizzato e inserito in pagina senza pericoli.

La CSP viene impostata con la direttiva *trusted-types 'myPolicy'*, che indica al browser di accettare solo le politiche Trusted Types nominate:

```
// Nell'header di risposta HTTP
Content-Security-Policy: default-src 'self' ; trusted-types 'myPolicy'
```

Nota: questa tecnologia è ancora sperimentale e non è ancora pienamente compatibile con tutti i browser moderni.

5.2 CONTROMISURE

La gestione degli accessi richiede una progettazione accurata, identificando le risorse, le criticità e definendo ruoli e politiche di accesso. È fondamentale che l'implementazione rispetti il design, evitando errori che possano compromettere la sicurezza, l'integrità, la riservatezza e la disponibilità dei dati. Le contromisure per una corretta progettazione e implementazione includono:

- Utilizzare modelli consolidati di controllo accessi adatti al contesto, spesso integrati in framework di sviluppo.
- Implementare controlli di autenticazione e autorizzazione per garantire che chi richiede accesso sia chi dice di essere e sia autorizzato.
- Utilizzare identificativi non predicibili (es. GUID) per ridurre l'esposizione di informazioni.
- Limitare l'accesso a componenti deprecate e vulnerabili.
- Non fare affidamento sulla Security through Obscurity, ma adottare un approccio di Defense in Depth.
- Implementare throttling e rate limit per le richieste.
- Assicurarsi della corretta configurazione dei controlli per le risorse cross-domain.

6 PRIVILEGE MANAGEMENT

Il **Privilege Management** è essenziale per la sicurezza informatica, gestendo i diritti di accesso all'interno di un'organizzazione, ed è parte dell'Access Control. Si basa su principi come il *Privilegio Minimo*, che concede agli utenti solo i diritti necessari e per il tempo strettamente necessario. È fondamentale monitorare in tempo reale per rilevare attività sospette. Una gestione efficace include la revisione periodic dei diritti, assicurando che siano adeguati. Soluzioni come *Privileged Identity Management (PIM)* e *Privileged Access Management (PAM)* aiutano ad automatizzare la gestione dei privilegi, limitando l'accesso elevato solo quando necessario. per il tempo necessario.

6.1 ESEMPI

6.1.1 Privilegi di Default

Anche quando implementati processi di gestione dei permessi ben strutturati, alcuni sistemi tendono a trascurare la rilevanza dei *Permessi di Default*. Questi consistono in tutta quella serie di autorizzazioni che un'utenza ha non appena viene creata e che possono essere, come spesso accade, troppo elevate rispetto allo scopo per la quale è stata creata l'utenza.

Scenario di Esempio I:

L'applicazione Web "*vulnerable.site*" prevede una fase di registrazione attraverso l'inserimento di una email per i suoi utenti. Ogni nuovo utente riceverà quindi una mail di conferma inviata da un amministratore per completare l'onboarding ed iniziare ad utilizzare i servizi messi a disposizione. Nell'implementazione dell'applicazione, ogni utenza è mappata con una classe del tipo:

```
class Utente{
      string username;          string password;
          int role;
}
```

Nel sistema descritto, ai nuovi utenti viene assegnato un valore di "role" pari a 0, che corrisponde al ruolo "Amministratore" nel database, fino alla conferma dell'email e al completamento del profilo. Questo crea una vulnerabilità, poiché gli utenti hanno privilegi amministrativi finché non completano la registrazione. Un attaccante potrebbe sfruttare questa situazione per compromettere il sistema. La soluzione consiste nell'assegnare privilegi minimi e espliciti di default a tutti i nuovi utenti.

6.1.2 Privilege Escalation Orizzontale e Privilege Escalation Verticale

La "**Privilege Escalation**" è un attacco in cui un malintenzionato ottiene accessi o privilegi non autorizzati sfruttando vulnerabilità di sicurezza. Gli attacchi possono essere di due tipi principali:

- **Orizzontale**: Sfrutta vulnerabilità nei controlli di accesso orizzontali, che regolano l'accesso a risorse dello stesso tipo per utenti con lo stesso livello di privilegi. Un esempio è quando un attaccante ottiene l'accesso ai dati di altri utenti dello stesso livello di privilegi, compromettendo la privacy dei dati.

- **Verticale**: Sfrutta vulnerabilità nei controlli di accesso verticali, che regolano l'accesso a risorse per utenti con diversi livelli di privilegi. Un esempio è quando un attaccante ottiene privilegi più elevati (es. amministrativi), compromettendo la sicurezza del sistema.

Entrambi i tipi di attacco derivano dal fallimento nell'implementazione di meccanismi di controllo adeguati.

6.2 CONTROMISURE

La gestione dei permessi è cruciale per un efficace controllo degli accessi. Inizia con la definizione delle matrici di autorizzazione, che mappano chi pu accedere a quali risorse e con quali privilegi. Si identificano le risorse da proteggere, si definiscono i ruoli (ad esempio, amministratore, lettore) e si assegnano i livelli di accesso appropriati per ciascun ruolo. La matrice previene accessi non autorizzati e riduce il rischio di violazioni della sicurezza, limitando le azioni possibili a quanto autorizzato. Deve essere aggiornata periodicamente per mantenere la sicurezza. La matrice di autorizzazione viene spesso rappresentata in forma di tabella, con le risorse elencate in riga e i ruoli in colonna, o viceversa. Ogni cella della tabella indica il tipo di autorizzazione concessa: pu essere vuota (nessun accesso), contenere una "R" (sola lettura), una "W" (sola scrittura), una "RW" (lettura e scrittura), o altri codici a seconda delle convenzioni adottate dall'organizzazione. La matrice deve essere mantenuta aggiornata e riveduta periodicamente per riflettere eventuali cambiamenti nei ruoli o nelle risorse.

Ruolo/Risorse	Risorsa 1	Risorsa 2	Risorsa 3
Amministratore	RW	RW	RW
Moderatore	R	RW	RW
Visualizzatore		R	RW

La matrice di autorizzazione pu essere integrata con sistemi di controllo degli accessi automatizzati, che utilizzano la matrice per gestire l'accesso alle risorse in tempo reale e registrare tentativi di accesso, aumentando la sicurezza e la tracciabilità. Essa rappresenta un elemento chiave nella gestione della sicurezza e nella compliance con normative di settore.

Di seguito si riportano le contromisure e gli accorgimenti da adottare per la progettazione ed implementazione di meccanismi di gestione dei privilegi:

- Utilizzare framework consolidati e soluzioni di **Privileged Identity Management (PIM)** e **Privileged Access Management (PAM)** piuttosto che implementazioni personalizzate.
- Progettare dettagliatamente ruoli e privilegi con strumenti come le matrici di autorizzazione.
- Monitorare e aggiornare continuamente i ruoli per rispondere a nuove necessità di funzionalità e sicurezza.
- Definire correttamente i permessi di default per evitare vulnerabilità.
- Implementare meccanismi di controllo degli accessi verticali e orizzontali adeguati.

7 INPUT VALIDATION & SANITIZATION

La **validazione dell'input** è fondamentale per garantire la sicurezza e l'integrità dei dati in un'applicazione, evitando vulnerabilità o input dannosi. Essa include due strategie principali:

- **Validazione sintattica**: Verifica la correttezza della sintassi dei dati, come i caratteri speciali.
- **Validazione semantica**: Controlla la coerenza dei valori nel contesto (ad esempio, una data di inizio precedente alla data di fine).

È particolarmente importante nei form web, dove un attaccante potrebbe sfruttare l'input non validato per iniettare codice dannoso. La validazione previene attacchi come SQL injection e XSS. Insieme alla validazione, si utilizza la **sanitizzazione**, che ripulisce l'input rimuovendo o neutralizzando caratteri pericolosi, impedendo che vengano interpretati come codice eseguibile. Entrambe le tecniche mirano a prevenire attacchi, con la validazione che verifica la correttezza dei dati e la sanitizzazione che li ripulisce.

7.1 ESEMPI

7.1.1 Command Injection

Il termine "**injection**" si riferisce a una classe di attacchi in cui un attaccante inietta codice dannoso in un'applicazione o sistema, che poi viene eseguito o processato. Questo pu causare accessi non autorizzati, corruzione dei dati e malfunzionamenti. Un esempio comune è la **Command Injection**, in cui l'attaccante esegue comandi arbitrari sul sistema operativo host attraverso un'applicazione vulnerabile. Ci accade quando l'applicazione invia dati non validati (come moduli, cookie o intestazioni HTTP) a una shell di sistema.

```
# Assume che abbiamo una funzione che esegue un comando di sistema
# e restituisce l'output (ad esempio, in Python, potrebbe essere os.system() o subprocess.run()) funzione
esegui_comando(comando):
    output = esegui_comando_di_sistema(comando)
restituisci output
# Input dell'utente (ad esempio, da un modulo web o da un argomento della riga di comando) input_utente
= ottieni_input_utente()
# Costruisci un comando utilizzando l'input dell'utente
# In questo esempio, supponiamo che l'input dell'utente sia un nome di file
# e vogliamo visualizzarne il contenuto comando
= "cat " + input_utente
# Esegui il comando
risultato = esegui_comando(comando)
# Mostra il risultato all'utente stampa("Output
del comando:") stampa(risultato)
```

7.1.2 SQLi (SQL Injection)

Le **SQL Injection** sono un tipo di attacco informatico che sfrutta le vulnerabilità nella gestione delle query SQL. Un attaccante fornisce in input del codice SQL dannoso (ad esempio, in un form web), affinché tale codice verrà poi eseguito sul database come parte della query. Questo pu portare a comportamenti indesiderati, come l'accesso non autorizzato ai dati, la corruzione di quest'ultimi o di intere tabelle. Esempio di codice vulnerabile a SQL Injection

```
funzione login(username, password) {
  // Concatenazione diretta dell'input dell'utente in una query SQL
  query = "SELECT * FROM users WHERE username = '" + username + "' AND password = '" + password + "'"
risultato = eseguiQuery(query)

  // ...
```

```
}
```

In questo esempio, un attaccante potrebbe inserire un valore come ***' OR '1'='1'; --*** per il nome utente, che porterebbe alla query *SQL:*

SELECT * FROM users WHERE username = ' ' OR '1'='1'; --' AND password = 'qualunquecosa'

permettendo all'attaccante di accedere ai dati contenuti nella tabella *users*, la verifica della password non viene eseguita.

7.1.3 Cross-Site Scripting (XSS)

Gli attacchi **Cross-Site Scripting (XSS)** sfruttano quelle vulnerabilità che permettono agli attaccanti di iniettare script dannosi in una pagina web legittima o in un'applicazione web. L'attacco avviene ai danni degli utenti che visiteranno la pagina o l'applicazione e che eseguiranno quello script con il codice malevolo. Questi script possono accedere a informazioni sensibili, come i cookie di sessione dell'utente.

```
funzione mostraCommento(commento) {
  // Inserimento diretto del commento in una pagina web senza escape dei caratteri speciali
paginaWeb.inserisciHTML("<p>" + commento + "</p>")
}
```

In questo esempio, un attaccante potrebbe inserire un valore come <script>/* codice dannoso */</script> per il commento, che porterebbe all'esecuzione del codice JavaScript dannoso nel browser degli utenti quando visualizzano la pagina.

7.1.4 Buffer Overflow

Un **Buffer Overflow** si verifica quando un programma riceve più dati di quanto la sua memoria possa contenere, causando il traboccamento di informazioni in aree di memoria adiacenti. Questo pu corrompere o sovrascrivere i dati, generando comportamenti anomali. Se un attaccante conosce il layout di memoria del programma, pu sfruttare questa vulnerabilità per inserire dati malevoli, sovrascrivendo la memoria contenente il codice eseguibile e prendendo il controllo del sistema.

```
char buf[64]; gets(buf);
```

7.2 CONTROMISURE

Di seguito si riportano le contromisure e gli accorgimenti da adottare per evitare i suddetti attacchi e per implementare delle tecniche di validazione e sanitizzazione efficaci:

- **Whitelist:** Utilizzare whitelist per definire i caratteri accettati in input, preferibile rispetto alle blacklist, che possono essere aggirate. Le whitelist possono essere implementate tramite espressioni regolari per accettare solo determinati pattern di caratteri.
- **Sanitizzazione**: Effettuare l'escape dei caratteri speciali (ad esempio, & con &amp), per prevenire attacchi di tipo Injection e XSS, impedendo l'esecuzione di script malevoli.

- **Stored Procedures o Query Parametriche**: Usare stored procedures o query parametriche quando si interagisce con il database per validare l'input dell'utente prima che venga processato, proteggendo contro le SQL Injection.
- **Corretta esecuzione dei comandi sul sistema operativo**: Evitare l'esecuzione di comandi di sistema con input dell'utente, eseguendo sempre la validazione e sanitizzazione dell'input.
- **Prevenzione dell'overflow**: Verificare che l'input dell'utente non superi la lunghezza del buffer, utilizzando funzioni che limitano l'input e generano eccezioni se si tenta di scrivere oltre il limite. Implementare tecniche come Address Space Layout Randomization (ASLR) e Data Execution Prevention (DEP).

8 GESTIONE DELLA CRITTOGRAFIA

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

- **Complessità**: La crittografia richiede una conoscenza approfondita dei principi matematici, e la sua complessità pu portare a errori che gli aggressori potrebbero sfruttare.
- **Gestione delle chiavi**: La sicurezza dei dati crittografati dipende dalla gestione sicura delle chiavi, inclusi la loro creazione, distribuzione, archiviazione e distruzione.
- **Crittografia obsoleta**: Algoritmi crittografici obsoleti possono diventare vulnerabili con l'avanzare della tecnologia, quindi è essenziale mantenere i sistemi crittografici aggiornati.
- **Attacchi di forza bruta**: Gli attacchi di forza bruta, che tentano tutte le combinazioni di chiavi, sono una minaccia, e i computer quantistici potrebbero renderli più fattibili.
- **Equilibrio tra sicurezza e prestazioni**: La crittografia pu rallentare i sistemi e le reti, richiedendo un giusto equilibrio tra sicurezza e prestazioni.
- **Crittografia end-to-end**: Sebbene offra maggiore sicurezza, pu ostacolare la rilevazione di malware e altre minacce nei dati crittografati.
- **Conformità normativa**: Le normative sulla privacy e sulla sicurezza dei dati influenzano l'uso della crittografia e devono essere rispettate dalle organizzazioni.
- **Accesso legittimo ai dati**: Le organizzazioni o le autorità possono necessitare l'accesso ai dati crittografati, sollevando il dilemma tra privacy e sicurezza.
- **Formazione degli utenti**: Gli utenti devono essere formati per comprendere e utilizzare correttamente la crittografia.

Affrontare queste sfide richiede un approccio olistico che integri tecnologia, formazione e politiche efficaci.

8.1 ESEMPI

8.1.1 Gestione delle chiavi

Questo esempio mostra come la mancanza di sicurezza nella gestione delle chiavi pu portare a vulnerabilità:

```
funzione GestioneChiaviNonSicura()
// Generazione della chiave     chiave
= GeneraChiave()

    // Stampa la chiave in chiaro (NON sicuro)
    Stampa("Chiave:", chiave)

    // Creazione di un'istanza di cifratura utilizzando la chiave
cifratura = CreaCifratura(chiave)

    // Il nostro messaggio originale
messaggio_originale = "Ciao Mondo"

    // Cifratura del messaggio
    messaggio_cifrato = cifratura.Cifra(messaggio_originale)
Stampa("Messaggio cifrato:", messaggio_cifrato)

    // Decifratura del messaggio
    messaggio_decifrato = cifratura.Decifra(messaggio_cifrato)
Stampa("Messaggio decifrato:", messaggio_decifrato)
}
```

In questo esempio, la chiave viene generata utilizzando GeneraChiave(), ma poi viene stampata in chiaro. Questo è un esempio di pratica non sicura, poiché esporre la chiave in questo modo la rende accessibile a chiunque abbia accesso all'output del programma. In un contesto reale, la chiave dovrebbe essere conservata in modo sicuro e non dovrebbe essere esposta o trasmessa in modo insicuro.

8.1.2 Crittografia obsoleta

Questo esempio mostra come un algoritmo di crittografia deprecato, come DES (Data Encryption Standard), pu essere facilmente compromesso.

```
funzione CrittografiaObsoleta()
    // Generazione della chiave utilizzando DES (NON sicuro)     chiave = GeneraChiaveDES()

    // Creazione di un'istanza di cifratura DES utilizzando la chiave     cifratura = CreaCifraturaDES(chiave)

    // Il nostro messaggio originale
    messaggio_originale = "Ciao Mondo"

    // Cifratura del messaggio utilizzando DES
    messaggio_cifrato = cifratura.Cifra(messaggio_originale)
    Stampa("Messaggio cifrato:", messaggio_cifrato)

    // Decifratura del messaggio utilizzando DES
    messaggio_decifrato = cifratura.Decifra(messaggio_cifrato)
    Stampa("Messaggio decifrato:", messaggio_decifrato)
}
```

In questo esempio, viene utilizzato l'algoritmo DES per cifrare e decifrare un messaggio. Tuttavia, DES è considerato obsoleto e non sicuro, poiché pu essere facilmente compromesso con attacchi di forza bruta. In un contesto reale, bisogna utilizzare algoritmi di crittografia più sicuri e aggiornati, come AES (Advanced Encryption Standard).

8.2 CONTROMISURE

Di seguito si riportano gli accorgimenti da seguire per evitare problematiche afferenti all'uso di crittografia debole:

- **Utilizzo di algoritmi di crittografia forti**: scegliere algoritmi di crittografia riconosciuti e testati come AES o RSA, per proteggere le chiavi ed eventuali dati *at rest* ed evitare l'uso di algoritmi di crittografia custom;
- **Gestione delle chiavi di crittografia**: assicurarsi di avere un sistema sicuro per la generazione, lo scambio, lo stoccaggio e la distruzione delle chiavi di crittografia;
- **Protocolli di trasmissione sicuri**: implementare la crittografia end-to-end per garantire che i dati siano protetti in fase di transito e utilizzare protocolli basati su TLS per proteggerli;

9 CONFIGURAZIONI INSICURE

Una **Configurazione insicura** si verifica quando le impostazioni di sicurezza di un'applicazione non sono definite in modo adeguato o sono implementate in modo errato. Questo pu includere l'utilizzo di impostazioni predefinite (in altri termini, la mancata modifica delle configurazioni di default), mal configurate o funzionalità non necessarie. Queste *misconfigurations* possono portare ad accessi non autorizzati, violazioni dei dati, compromissione dell'applicazione e altri incidenti di sicurezza.

9.1 ESEMPI

Di seguito si riportano alcuni esempi di configurazioni poco sicure:

- **Credenziali di default**: Le applicazioni spesso vengono configurate con credenziali di accesso predefinite (es. "admin"/"admin", "1234"). Se non vengono modificate durante la configurazione, gli attaccanti possono facilmente ottenere l'accesso.
- **Configurazione dei permessi**: Applicazioni mobili che richiedono l'accesso a risorse non necessarie, come una app per appunti che chiede l'accesso ai contatti o alla posizione, rappresentano una configurazione errata e un rischio di sicurezza.
- **Funzionalità non necessarie**: Non disabilitare le funzionalità non essenziali, come pagine web o porte aperte non utilizzate, pu compromettere la sicurezza dell'applicazione.
- **Elenco delle directory abilitato**: Un web server che consente agli utenti di visualizzare l'elenco delle directory rappresenta un rischio di sicurezza, poiché potrebbe permettere a un attaccante di accedere a file sensibili e violare i dati.
- **Protocolli e crittografia obsoleti**: L'accesso a un sito web senza crittografia (es. HTTP invece di HTTPS) rende vulnerabili i dati, che vengono trasmessi in chiaro e possono essere facilmente intercettati da malintenzionati**.**

```
public class RedirectExample {
    public static void main(String[] args) {
        // Supponiamo che tu abbia un oggetto HttpServletResponse
        // disponibile nel contesto della tua applicazione web
        HttpServletResponse response = ...; // Inizializza l'oggetto response

        String urlRedirect = "http://nuovo-esempio.com";

        try {
            // Imposta il codice di stato 302 (Found) per il redirect
response.setStatus(HttpServletResponse.SC_FOUND);              //
Imposta l'header "Location" con l'URL di destinazione
response.setHeader("Location", urlRedirect);
            // Invia la risposta al client
response.flushBuffer();          }
catch (IOException e) {
e.printStackTrace();
        }
    }
}
```

- **Gestione dei cookies**: quando il server che crea e invia un cookie non è configurato con il flag "*secure*". Quest'ultima è di fondamentale importanza, in quanto, senza di essa i cookies verrebbero trasmessi in chiaro, tramite una connessione non crittografata.

```
funzione gestisciRichiesta(richiesta, risposta) {
// Creazione di un cookie senza la flag "secure"
  cookie = creaCookie("nomeUtente", "Mario", "secure" = FALSO)

  // Aggiunta del cookie all'header della risposta   risposta.aggiungiHeader("Set-
Cookie", cookie)

  // Invio della risposta al client
risposta.invia("Ciao mondo!") }
```

9.2 CONTROMISURE

Di seguito si riportano i suggerimenti per prevenire i casi più comuni di misconfiguration di sicurezza:

- **Modificare le configurazioni di default**: l'utilizzo di credenziali, certificati o configurazioni predefinite pu esporre vulnerabilità;
- **Non integrare nel codice protocolli e/o cifratura deprecati**: utilizzare protocolli di comunicazione sicuri, come HTTPS e assicurarsi di utilizzare algoritmi di cifratura standard e sicuri anche per i dati *at rest*;
- **"Pulire" l'applicazione da quanto non essenziale per il suo funzionamento**: come parte del processo di configurazione di un applicativo, è necessario rimuovere tutte le funzionalità non utilizzate, i commenti rimasti nel codice e i messaggi di errore che rivelano informazioni sensibili al fine di rendere l'applicazione una piattaforma quanto più minimale possibile;
- **Evitare accessi non autorizzati**: verificare che le impostazioni configurate non consentano accesso a specifiche aree dell'applicazione agli utenti non autorizzati.
- **Conduzione di controlli frequenti**: si suggerisce di svolgere ispezioni periodiche per individuare e ridurre le potenziali configurazioni errate della sicurezza e i rischi per la sicurezza delle applicazioni.

10 UTILIZZO DI CODICE DI TERZE PARTI

Durante lo sviluppo delle applicazioni, le librerie esterne di terze parti vengono frequentemente utilizzate per risparmiare tempo e sforzi. Tuttavia, queste librerie potrebbero contenere difetti o vulnerabilità di sicurezza sfruttabili da attaccanti, specialmente se sono obsolete o non supportano più aggiornamenti. In tal caso, le vulnerabilità della libreria si riflettono nell'intera applicazione.

Il processo di **Vulnerability Management** è essenziale per identificare, valutare e mitigare le vulnerabilità. Si sviluppa in quattro fasi:

- **Identificazione delle vulnerabilità**: Utilizzo di scanner e strumenti per individuare debolezze nei sistemi.
- **Valutazione delle vulnerabilità**: Analisi delle vulnerabilità per determinarne gravità e impatto sulla sicurezza.
- **Trattamento delle vulnerabilità**: Correzione del codice, applicazione di patch e mitigazione del rischio.
- **Segnalazione delle vulnerabilità**: Documentazione delle vulnerabilità, delle azioni intraprese e dei risultati ottenuti.

Il processo è ciclico e richiede monitoraggio continuo per garantire che le vulnerabilità vengano gestite tempestivamente.

La riduzione della superficie di attacco consiste nel minimizzare i punti vulnerabili eliminando componenti, librerie e codice non utilizzato, riducendo i rischi di sicurezza. Una superficie di attacco più piccola facilita il monitoraggio e la difesa, permettendo di concentrarsi sui punti critici. È essenziale valutare continuamente e aggiornare le componenti software per mantenere un ambiente sicuro.

10.1 ESEMPI

Un esempio di utilizzo di libreria di terze parti vulnerabile è jQuery in una delle versioni datate afflitte da vulnerabilità (in questo caso 1.9.1). La libreria, se non aggiornata, porta ad esposizione verso attacchi di Cross Site Scripting (XSS).



10.2 CONTROMISURE

Di seguito si riportano i suggerimenti per prevenire problematiche afferenti all'utilizzo di librerie di terze parti:

- **Aggiornamento delle librerie**: È fondamentale mantenere aggiornate le librerie, garantendo che le versioni utilizzate non siano obsolete o vulnerabili.
- **Monitoraggio delle librerie dipendenti**: Le librerie che utilizzano altre librerie di terze parti (dipendenze) devono essere monitorate. Se una dipendenza presenta vulnerabilità, è consigliato aggiornare la libreria principale a una versione più sicura o sostituirla con una libreria alternativa senza dipendenze vulnerabili.

- **Analisi SCA (Software Composition Analysis)**: Si suggerisce di eseguire analisi SCA utilizzando strumenti automatici per identificare librerie e dipendenze vulnerabili nel codice, facilitando la gestione della sicurezza.

10.3 LOGGING E MONITORAGGIO

Le vulnerabilità inerenti al deficit di **Logging e Monitoraggio** si verificano quando un'applicazione o un sistema non registra e monitora adeguatamente gli eventi, rendendo difficile rilevare e rispondere agli incidenti di sicurezza. Ci pu consentire agli attaccanti di eseguire attacchi senza essere rilevati e pu anche ostacolare gli interventi di risposta agli incidenti. Nella pratica, ci accade solitamente quando:

- Eventi su cui è necessario fare audit, come i tentativi di accesso, i login falliti e le transazioni di alto valore, non vengono registrati.
- Avvisi ed errori generano messaggi di log poco descrittivi o poco chiari.
- I log delle applicazioni e delle API non vengono monitorati per attività sospette.
- I log vengono archiviati solo localmente.
- Soglie di allarme adeguate e processi di escalation della risposta non sono in atto o non sono efficaci.
- L'applicazione non è in grado di rilevare, segnalare o rispondere agli attacchi attivi in tempo reale o quasi in tempo reale.

10.4 ESEMPI

Nel seguente esempio, il codice tenta di autenticare l'utente: se il login fallisce, viene effettuato un nuovo tentativo. In questo caso, il login fallito non viene registrato e non ci sarebbe alcuna traccia di un eventuale tentativo di brute force da parte di un avversario.

```
if LoginUser(){ // Login successful
RunProgram();
} else {
// Login unsuccessful LoginRetry();
}
```

10.5 CONTROMISURE

Di seguito si riportano i suggerimenti per prevenire problematiche afferenti ad un'insufficienza dei meccanismi di logging e di monitoraggio:

- **Logging adeguato**: assicurarsi che eventi, come i tentativi di accesso, l'esecuzione di chiamate alle API applicative e, in generale, le transazioni più significative, vengano registrati. È necessario generare dei log che siano chiari e che illustrino esaustivamente ci che li ha generati, specificando utente, motivazione e timestamp. SI suggerisce di utilizzare formati interpretabili da strumenti automatici di raccolta dei log.
- **Conservazione dei log**: è fondamentale che l'accesso ai file di log sia protetto e consentito solo agli utenti autorizzati. Inoltre, come da legislazione i log vanno hanno un periodo di "retention" ben definito e dipendente dalla tipologia di log.
- **Monitoraggio efficace**: si suggerisce di definire processi di monitoraggio delle applicazioni (esistono anche delle soluzioni automatiche), configurando meccanismi di *alerting* e di notifica in caso di comportamento imprevisto e/o sospetto.

11 ELENCO ALLEGATI

Allegato		Descrizione
I	Hardening	Illustra i principi e le tecniche del processo di Hardening da attuare su specifici sistemi ed applicativi, con l'obiettivo di incrementarne e rafforzarne la sicurezza.

2	JavaScript	Illustra gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo nella tecnologia JavaScript. 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.
3	NodeJS	Questo documento ha lo scopo di illustrare gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo nella tecnologia NodeJS. 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.
4	ABAP	Questo documento ha lo scopo di illustrare gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo nella tecnologia ABAP 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.
5	.NET	Questo documento ha lo scopo di illustrare gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo nella tecnologia .NET 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.
6	Angular	Questo documento ha lo scopo di illustrare gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo nella tecnologia Angular. 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.
7	C/C++	Questo documento ha lo scopo di illustrare gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo nella tecnologia C/C++. 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.
8	Flutter	Questo documento ha lo scopo di illustrare gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo nella tecnologia Flutter. 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.
9	Java	Questo documento ha lo scopo di illustrare gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo nella tecnologia Java. 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.
10	Link utili	Contiene link utili e riferimenti a documenti utilizzati a supporto.