Gestione sicura delle credenziali e dei segreti in memoria

Pasquale Ardimento, 2025

Sicurezza nelle Applicazioni

Obiettivi della presentazione

- Fornire una definizione di dati sensibili
- Spiegare perché dati sensibili possono rimanere in memoria e rischiare l'esfiltrazione
- Mostrare esempi pratici di codice vulnerabile e di attacco (memory dump)
- Presentare codice di mitigazione e best practice
- Fornire note operative e riferimenti agli strumenti (jmap, jhat, Eclipse MAT)
- Gestione dei segreti: Vault

Dati sensibili

I dati sensibili in un'applicazione sono tutte le informazioni il cui accesso non autorizzato può causare danno agli utenti, all'organizzazione o alla riservatezza dell'ambiente. Di conseguenza questi dati richiedono controlli di sicurezza più stringenti.

Esempi

- **Credenziali di accesso**: password, PIN, OTP, token di sessione (es. session_id, JWT).
- **Chiavi e segreti**: chiavi private, segreti API, secret_id/role_id, token di servizio, KMS/HSM keys.
- **Dati personali identificabili** (PII): nome + identificativo (es. codice fiscale, ID), indirizzo, email, numero di telefono, data di nascita.
- **Categorie speciali / sensibili**: dati sanitari, informazioni sulla razza/origine etnica, opinioni politiche, dati biometrici (quando applicabile — richiedono protezioni legali).
- **Informazioni finanziarie**: numeri di carta di credito, dettagli bancari, transazioni sensibili.
- **Metadati e derivati**: log che contengono token, stack trace con segreti, file HPROF/dump contenenti credenziali in chiaro.
- **Configurazioni e certificati**: file di configurazione con segreti hard-coded, keystore, file PEM non protetti.

Raccomandazione di sicurezza per i dati sensibili

- **Non memorizzare in chiaro**: cifrare a riposo (memorizzati sul disco) e in transito sulla rete (TLS + crittografia a riposo).
- **Minimizzare la persistenza**: usare segreti temporanei e pulizia in memoria (buffer mutabili).
- **Centralizzare la gestione**: usare secret manager (Vault, cloud KMS) e non segreti hard-coded.
- **Least privilege & auditing**: *policies* granulari e audit log per tracciare accessi e operazioni.
- **Proteggere i dump**: considerare file HPROF, log e backup come dati sensibili.

TLS (Transport Layer Security)

- protocollo che protegge **le comunicazioni sulla rete**, assicurando che ciò che invii e ricevi tra client e server sia **cifrato**, **integro** e che il server (e opzionalmente il client) sia **autentico**.

Problema dei dati sensibili in Java

- Perché le String in Java sono immutabili:
  - Immutabilità per sicurezza e condivisione: gli oggetti String possono essere condivisi senza sincronizzazione, consentendo caching (string pool) e migliori prestazioni multi-thread.
  - Progettazione: la classe String è stata resa immutabile per semplificare l'uso come chiavi in strutture dati (hashing stabile) e per evitare problemi di concorrenza.
- Cosa significa in pratica:
  - una String non può essere modificata dopo la creazione; ogni operazione che sembra cambiarla crea un nuovo oggetto String.
- Cos'è un memory dump:
  - Un memory dump (heap dump) è un'istantanea della memoria heap del processo (JVM) in un dato momento. Contiene gli oggetti, i loro campi e i riferimenti.
  - Viene generato con strumenti (es. jmap) e salvato in un file (HPROF) per l'analisi offline con tool come jhat o Eclipse MAT.
- Pericolo pratico:
  - se le password sono memorizzate in oggetti immutabili (String), i loro valori possono rimanere nell'heap fino a quando non vengono rimossi dal GC; un dump dell'heap può rivelare quei valori.

Esempio di vulnerabilità

```
// Classe vulnerabile: Password.java
public class Password {
    private String username;
    private String password;

    public static void main(String[] args) {
        Console c = System.console();
        Password pwd = new Password();
        pwd.setUsername(c.readLine("Inserire user name: "));
        pwd.setPassword(c.readLine("Inserire password: "));
        // la password è ora in memoria come String immutabile
        System.out.println("Attenzione: credenziali esposte in memoria fino al GC.");
    }
    // getter / setter ...
}
```

Scenario di attacco: Memory Dump

- **Prerequisiti per l'attaccante**:
  - Accesso alla macchina host (locale o remoto) con privilegi sufficienti per eseguire comandi sul processo Java (spesso lo stesso utente o root).
  - L'applicazione deve essere in esecuzione (jmap opera su processi JVM attivi) o il file HPROF dev'essere disponibile.
- **Come l'attaccante può ottenere accesso**:
  - accesso via SSH sfruttando credenziali compromesse o vulnerabilità di rete; presenza di utenti malintenzionati con accesso fisico; esecuzione di codice malevole che apre una shell.
- **Esecuzione del dump e analisi:**
  - **jps**: elenca i processi Java attivi e mostra il PID.
  - **jmap** -dump:live,format=b,file=memory_dump.hprof <PID>: crea un file HPROF contenente l'heap (opzione live salva solo oggetti raggiungibili).
  - **jhat** memory_dump.hprof: avvia un server web che permette di esplorare l'HPROF e usare OQL per query sugli oggetti.
  - Se jhat non fosse disponibile (compatibile fino a jdk 1.8) si usi VisualVM
    - brew install --cask visualvm
    - open -a VisualVM
- **Note di sicurezza**: l'uso di jmap/jhat/visualVM può richiedere privilegi e può impattare le prestazioni del sistema.

Che cos'è HPROF (heap profile)

- HPROF è il formato binario usato da Java per salvare un heap dump (profilo della heap).
- Viene prodotto, ad esempio, con:
  - - jmap -dump:live,format=b,file=memory_dump.hprof <PID>
- Contenuto di un file HPROF:
  - header, tabella degli oggetti, array di byte per le stringhe, riferimenti e metadati.
- Uso: gli analisti caricano l'HPROF in strumenti di analisi (jhat, Eclipse MAT, VisualVM) per esplorare gli oggetti e trovare dati sensibili.
- **Nota**: un file HPROF è spesso abbastanza grande e può contenere informazioni sensibili in chiaro; va trattato con attenzione.

Scenario di attacco...

```
# 1) elencare processi Java e trovare PID
jps -l

# 2) dump della heap (HPROF)
jmap -dump:live,format=b,file=memory_dump.hprof <PID>

# 3) analisi con jhat (apre server web su 7000)
jhat memory_dump.hprof

# 3) in alternativa analisi con visualVM
open -a VisualVM

# 3) in alternativa analisi con Eclipse MAT per analisi offline
```

...Scenario di attacco

```
# Flusso concettuale
# 1) usare OQL su jhat: select p from esempioPwd.Password p
# 2) selezionare l'istanza e leggere il campo 'password' come bytes esadecimali
# 3) convertire i byte in stringa (HexToString)
# 4) ottenere la password in chiaro
```

Comandi e opzioni: jps, jmap, jhat (spiegazione)

```
jps -l
 - Elenca i processi Java in esecuzione con il PID e il main class/JAR.

jmap -dump:live,format=b,file=memory_dump.hprof <PID>
 - dump: opzione per creare un heap dump.
 - live: include solo gli oggetti raggiungibili (riduce rumore).
 - format=b: esporta in formato binario HPROF.
 - file=...: percorso del file di output.

jhat memory_dump.hprof
 - Avvia un server web (di default porta 7000) per esplorare il dump
HPROF.
 - Fornisce console OQL (Object Query Language) per interrogare oggetti
JVM.
Oppure open -a VisualVM
Apri il file

Permessi e limiti:
 - jmap potrebbe richiedere lo stesso utente di esecuzione del processo
Java o privilegi di root.
 - Generare dump su sistemi in produzione può avere impatto sulle
prestazioni; usare con cautela.
```

Dati sensibili: Mitigazione a livello di codice

```
// SecurePassword.java - mitigazione
import java.io.Console;
import java.util.Arrays;

public class SecurePassword {
    private String username;
    private char[] password;

    public static void main(String[] args) {
        Console c = System.console();
        SecurePassword sp = new SecurePassword();
        sp.setUsername(c.readLine("Inserire user name: "));
        sp.setPassword(c.readLine("Inserire password: ").toCharArray());

        // uso della password...
        sp.clearPassword(); // pulizia immediata
    }

    public void setPassword(char[] pwd) { this.password = pwd; }
    public void clearPassword() { Arrays.fill(this.password, '\0'); }
}
```

Dati sensibili: Mitigazione a livello di codice

```
// SecureFileReader.java
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;

public class SecureFileReader {
    private char[] data;

    public SecureFileReader(String file) throws IOException {
        data = new char[256];
        try (BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream(file)))) {
            br.read(data, 0, data.length);
        }
    }

    public void clearData() {
        java.util.Arrays.fill(data, '\0');
    }
}
```

Ulteriori mitigazioni e best practice

- Evitare logging di segreti e dump di eccezioni che rivelano dati sensibili.
  - logger.warn("Login failed for user={}", user); // no password
  - logger.error("Errore pagamento orderId={}, customerId={}", orderId, customerId, sanitizeException(e));

  sanitizeException(e) rimuove o anonimizza campi sensibili nel messaggio/stacktrace prima del log.
- Limitare privilegi e accesso a strumenti di diagnostica su server di produzione.
  - Limitare accesso ai server di produzione e agli strumenti (jmap, jstack, ...). Disabilita i servizi di debug in produzione e usa controllo degli accessi centralizzato (MFA, ruoli).
    Esempio operativo:
    - Crea gruppo ops e aggiungi solo gli operatori autorizzati.
    - Sudoers (esempio): %ops ALL=(root) /usr/bin/systemctl restart myapp (consente solo il comando necessario).
    - Disabilita porte JDWP / agenti di debug nelle JVM di produzione (-agentlib:jdwp=... rimosso).
- Usare secret manager per gestione centralizzata dei segreti (es. Vault).
- Formare gli sviluppatori sul ciclo di vita dei segreti in memoria.

Bloccare i dump mediante JVM

- La JVM può scrivere heap dump o thread dump automaticamente o quando qualcuno si attacca via jcmd, jmap, JMX ecc.
- È possibile ridurre il rischio con opzioni a runtime disabilitando i dump automatici su OOM (OutOfMemoryError):
  - java -XX:-HeapDumpOnOutOfMemoryError -jar app.jar
- Se non si vuole che altri processi si attacchino alla JVM (con jmap, jcmd, ecc.):
  - java -XX:+DisableAttachMechanism -jar app.jar

Cos’è Vault

- Vault è una soluzione per la gestione centralizzata dei segreti (password, token, chiavi, certificati).
  - https://www.hashicorp.com/en/products/vault
- Fornisce controllo accessi basato su identità, auditing, e API per integrazione con app e servizi.
- Fonte: documentazione ufficiale HashiCorp Vault.

Server Vault & Storage backend

- Server Vault
  - Espone API REST, applica policy, coordina secrets engines e gestisce l’autenticazione.
  - Tratta le richieste ma non “conserva” necessariamente i segreti in memoria permanente: li persiste nello *storage backend*.
- Storage backend (ruolo)
  - Persistenza duratura dei dati di Vault (metadati, versioni segreti, token leasing).
  - Scelta del backend determina alta disponibilità, resilienza e requisiti operativi.
- Esempi comuni
  - Consul: storage distribuito, usato per HA.
  - File (local): solo per test/dev.
  - S3: storage cloud (AWS).
  - Integrated Storage (Raft): storage interno replicato, consigliato per HA senza dipendenze esterne.

Perché Vault è come una cassaforte della scuola

- Immagina che ogni bambino abbia le sue caramelle nello **zaino**.
- Se ognuno mette le caramelle nello zaino, qualcun altro congegno male può **rubare lo zaino** (o aprirlo se trova la chiave) e prendere le caramelle.
- Invece, la scuola mette tutte le caramelle **in una cassaforte del preside**.
- I bambini non tengono caramelle negli zaini: quando servono, chiedono alla cassaforte.
- La cassaforte dà **solo** la quantità necessaria e **solo per poco tempo** (cioè la password è temporanea).
- La cassaforte tiene un registro: chi ha chiesto cosa e quando.
- Se qualcuno ruba una singola caramella presa in prestito, non può usarla dopo poco: **scade** (diventa inutile).
- Quindi: meno caramelle sparse = meno rischio; la cassaforte controlla accessi, dà cose temporanee e registra tutto.

Perché Vault protegge i segreti — analogia pratica

- **Senza Vault**: ogni applicazione/istanza tende a memorizzare localmente le proprie credenziali operative (es. credenziali DB, API key, token, certificati TLS) in file di configurazione, variabili d'ambiente o database. È come avere **molti zaini** che contengono chiavi importanti.
  - Se uno zaino viene rubato o il server compromesso, il ladro ottiene i segreti.
- **Con Vault**: c'è una **cassaforte centralizzata** (Vault) che custodisce i segreti.
  - Le applicazioni **non** memorizzano più i segreti a lungo: chiedono a Vault al bisogno.
  - Vault può emettere **segreti temporanei** (credenziali che scadono automaticamente) e **segregare privilegi** con policy.
  - Tutte le richieste sono tracciate (audit), quindi si sa **chi** ha chiesto **cosa** e **quando**.

Perché Vault protegge i segreti — analogia pratica

- Benefici chiave: meno “secret sprawl”, rotazione automatica, auditing e controllo fine-grained.
  - **Meno *secret sprawl*** — meno segreti sparsi e duplicati in giro.
  - **Rotazione automatica** — segreti che vengono creati/ruotati automaticamente e scadono.
    - Vault può **generare credenziali temporanee** (es. username/password per DB) con scadenza (TTL). Alla scadenza il segreto non funziona più.
  - **Auditing e controllo *fine-grained*** — registro di tutto e regole molto precise su chi può fare cosa.
- Limite importante: la cassaforte stessa va protetta (TLS, auto-unseal, backup, least-privilege su chi può accedervi).

Meno 'Secret Sprawl' — Concetto

- **Cosa**: ridurre la quantità di copie dei segreti (password, API key, certificati) che esistono in molti posti.
- **Perché importante**: più copie = più superfici d’attacco e più punti di fallimento (config file, repo, variabili d'ambiente).
- **Come Vault aiuta**: centralizza i segreti, restituisce solo ciò che serve al runtime e riduce la distribuzione di valori hard-coded.

Meno 'Secret Sprawl' — Esempi pratici e checklist

- **Esempio**: prima ogni server ha db.password nel file; dopo, i server chiedono a Vault la password al bisogno.
- **Checklist operativa**: rimuovere password dai repo; usare Vault KV per segreti non dinamici; usare Agent per retrieval locale.

```
# Prima (insicuro)
# /etc/app/config.yml
                                        db:
                                    password: "pa$$w0rd"


# Dopo (più sicuro)
# uso runtime: il servizio recupera la password da Vault
                        vault kv get -field=db.password secret/myapp
# vault kv get recupera un segreto dal Key Value Store
# -field=db.password restituisci solo il valore associato alla chiave db.password
# secret/myapp il percorso della «cassaforte»
```

Prima di Vault

- Si immagini di avere più server, e su ognuno di essi c’è un file di configurazione scritto così:

  /etc/app/config.yml

  db:

  password: "pa$$w0rd"

- In pratica:
  - La password è scritta in chiaro, come se fosse scritta su un post-it attaccato al computer.
  - Chiunque entri nel server o apra il repository può leggerla.
  - Se qualcuno copia il file, ha automaticamente la password.
  - Cambiare la password è difficile: si deve entrare su ogni server e aggiornare il file a mano.

Ora, invece, si tolgano la password dai file.

- I file di configurazione non contengono più la password vera.
- Quando il programma parte, **chiede a Vault** la password, solo nel momento in cui ne ha bisogno: *vault kv get -field=db.password secret/myapp*

Così:

- La password non è mai scritta “a penna” su disco o nei repository.
- Solo i servizi autorizzati possono chiederla a Vault.
- Vault tiene tutte le password in cassaforte e permette di cambiarle in un solo punto (non su cento server diversi).
- Si possono anche avere password dinamiche che cambiano spesso, riducendo il rischio che qualcuno le rubi e le usi a lungo.

- Checklist operativa per usare Vault in pratica:
  - Rimuovere password e segreti dai repository e dai file di configurazione.
  - Usare **Vault KV** (Key-Value store) per segreti statici come chiavi API o password fisse.
  - Usare **Vault Agent** per permettere ai servizi di recuperare in automatico i segreti in locale, senza dover scrivere comandi a mano.

Accedere a Vault tramite Java: Linee guida

- Leggi VAULT_ADDR e VAULT_TOKEN da variabili d'ambiente (mai hardcode).
- Richiesta a KV v2: GET /v1/secret/data/myapp.
- Evita le String per i segreti: usa byte[] / char[] e azzera subito.
- db.password = password dell'utente dell'app per il DB (es. app_user).

```
byte[] body = httpGetBytes(vaultUrl, vaultToken);          // risposta JSON come byte[]
char[] json = new String(body, UTF_8).toCharArray();       // parsing con char[]
Arrays.fill(body, (byte)0);                                // pulisco i byte non appena possibile

char[] dbPass = extractField(json, "db.password");         // estrazione minimale
Arrays.fill(json, '\0');                                   // azzero buffer JSON

// usa la password (es. costruisci DataSource), poi azzera
useDbPassword(new String(dbPass));                         // evitalo se possibile
Arrays.fill(dbPass, '\0');
```

Accedere a Vault tramite Java: Esempio

```
String addr  = System.getenv("VAULT_ADDR"); // variabile di ambiente
String token = System.getenv("VAULT_TOKEN"); // variabile di ambiente
byte[] body = httpGetBytes(vaultUrl, vaultToken);          // risposta JSON come byte[]
char[] json = new String(body, UTF_8).toCharArray();       // parsing con char[]
Arrays.fill(body, (byte)0);                                // cleaning dei byte non appena possibile

char[] dbPass = extractField(json, "db.password");         // estrazione minimale
Arrays.fill(json, '\0');                                   // cleaning del buffer JSON

// utilizzo della password (es. costruisci DataSource), cleaning della password
useDbPassword(new String(dbPass));                         // evitalo se possibile
Arrays.fill(dbPass, '\0');
```

Rotazione automatica — Concetto

- **Cosa**: generazione e uso di segreti temporanei con durata (TTL) e revoca automatica.
- **Perché utile**: se una credenziale viene compromessa, la finestra di abuso è limitata (blast radius ridotto).
- **Tipi**: credenziali DB temporanee, credenziali cloud temporanee (AWS/GCP), certificati con TTL.

Rotazione automatica — Esempio pratico e operazioni

- **Esempio**: Database engine genera username/password temporanei per un'app con lease 1h.
- **Best practice**: configurare ttl/default_ttl, monitorare lease, automatizzare il rinnovo o rilasciare le credenziali.

```
# Richiedere credenziali temporanee (Database engine)
vault read database/creds/readonly
# Output tipico:
# username: v_abc123
# password: p@ssw0rd
# lease_id: database/creds/readonly/abcd
# lease_duration: 1h

# Revoca manuale (se necessario)
vault lease revoke <lease_id>
```

Auditing e controllo fine-grained — Concetto

- **Auditing**: registrare chi ha effettuato quale operazione su quali segreti e quando (log per forensic e compliance).
- **Controllo fine-grained**: policies HCL che definiscono permessi precisi per path e azioni (read, list, create, delete).
- **Come si combinano**: policy restrittive + audit log permettono di limitare e rivelare accessi non autorizzati.

Auditing e controllo fine-grained — Esempi operativi

- Abilitare audit per inviare log a file o syslog e integrare con SIEM per alerting.
- Scrivere policy ristrette seguendo il principio least-privilege.

```
# Abilitare audit su file
vault audit enable file file_path=/var/log/vault_audit.log


# Policy esempio (my-policy.hcl)
# chi possiede una token/role che include la policy my-policy potrà
leggere i segreti e (in teoria) listare le chiavi all'interno di
secret/data/myapp/*
path "secret/data/myapp/*" {
  capabilities = ["read", "list"]
} vault policy write my-policy my-policy.hcl
```

Perché Vault protegge i segreti — esempio

```
# l'app si autentica (es. AppRole)
token = Vault.login(role_id, secret_id)

# chiede credenziali temporanee per il DB
credenziali = Vault.get("database/creds/readonly")
# usa le credenziali; dopo TTL scaduto le credenziali non valgono più
```

- role_id = identificatore della role (tipo "username").
- secret_id = credenziale segreta (tipo "password") usata insieme al role_id per ottenere un token Vault.
- **Protezione**: trasmettere solo su TLS; non salvare secret_id in codice/repò; preferire response-wrapping, uso di Vault Agent o auth methods integrati (Kubernetes/AWS) per ridurre l’esposizione.

Flusso tipico raccomandato nell’uso di Vault

1. Deploy Vault Agent sullo stesso VM/container della webapp.
2. Agent fa auto-auth (AppRole/K8s) e ottiene token.
3. Agent scrive token e/o segue un template che genera file di config con i segreti: es. /etc/myapp/secrets.json (mode 600).
4. La webapp legge /etc/myapp/secrets.json all’avvio (oppure rilegge periodicamente / riceve segnale), usa i valori.
5. Agent rinnova token e rigenera file; l’app può rileggere o usare watch.

Svantaggi e limiti

- Richiede competenze operative e infrastruttura dedicata (HA, backup).
  - servizio critico che va mantenuto attivo
  - Servizio che richiede nodi ridondanti e procedure di backup e di ripristino
- Configurazione errata può creare single point of failure;
  - Se Vault è mal configurato, tutta l'app può bloccarsi perché non riceve più i segreti.
- Curve di apprendimento e potenziali costi se si sceglie la versione gestita.
  - Vault ha concetti non banali: **secret engine, lease, token, policy, approle**… serve formazione per i team.
  - La versione open source è gratuita, ma comporta costi di gestione interni.

Modalità di installazione

- Download binario ufficiale (HashiCorp releases) o package manager (apt, brew).
- Container/Docker per ambienti di test; Helm chart per deploy in Kubernetes.
- Per produzione: storage backend resilienti (Consul, Raft), TLS e auto-unseal.

Appendice Vault

Quickstart: dev server e vari comandi utili

```
# Avvio rapido (DEV — NON in produzione)
vault server -dev -dev-root-token-id="root"

# Impostare VAULT_ADDR e autenticarsi
export VAULT_ADDR='http://127.0.0.1:8200'
vault login root

# Inizializzazione (produzione)
vault operator init -key-shares=1 -key-threshold=1
# Unseal (usare le unseal keys salvate)
vault operator unseal <unseal_key>

# Avvio in background con storage consul (esempio)
vault server -config=/etc/vault.hcl
```

Esempio: abilitare KV v2 e usare secret KV

```
# Abilita KV v2
vault secrets enable -path=secret kv-v2

# Scrivi un segreto
vault kv put secret/myapp db.username='app' db.password='s3cr3t'

# Leggi un segreto
vault kv get secret/myapp
```

Esempio: AppRole (autenticazione macchina)

```
# Abilita AppRole auth method
vault auth enable approle

# Crea policy e role
vault policy write my-policy - <<EOF
path "secret/data/myapp/*" {
  capabilities = ["read"]
}
EOF

vault write auth/approle/role/my-role token_policies="my-policy"
vault read auth/approle/role/my-role/role-id
vault write -f auth/approle/role/my-role/secret-id
```

Riferimenti

- OWASP: Sensitive Data Exposure
- CERT: Secure Coding Guidelines for Java
- Documentazione Oracle su String e GC
- Tool: jmap, jhat, Eclipse MAT, VisualVM