DELIVERING CODE IN PRODUCTION WITH DEBUG FEATURES ACTIVATED

Le funzioni di **debug** di un'applicazione consentono agli sviluppatori di trovare più facilmente i problemi attraverso il monitoraggio del comportamento del sistema. Un aggressore potrebbe sfruttare tali informazioni per ottenere informazioni sensibili sul sistema ed i suoi utenti, per poi attaccarlo.

Ad esempio, ciò può avvenire a causa della funziona printStackTrace(), la quale stampa un Throwable e la sua traccia di stack in System.Err. Per stampare le informazioni dei Throwable è consigliabile usare i logger.

Security Hotspot

```
try {
  /* ... */
} catch(Exception e) {
  e.printStackTrace(); // Sensitive
}
```

Risoluzione

```
try {
  /* ... */
} catch(Exception e) {
  LOGGER.log("context", e);
}
```

FORMATTING SQL QUERIES

Le query SQL formattate attraverso la **concatenazione di più stringhe** possono portare ad SQL injection.

Quando una query è costruita attraverso la concatenazione di valori inseriti in input dagli utenti (es. campi di un form), è necessario disporre query parametriche attraverso PreparedStatement e "caricare" i relativi parametri.

Security Hotspot

```
Statement stmt2 = con.createStatement();

ResultSet rs2 = stmt2.executeQuery("select FNAME, LNAME, SSN
" + "from USERS where UNAME=" + user); // Sensitive

PreparedStatement pstmt = con.prepareStatement("select
FNAME, LNAME, SSN " + "from USERS where UNAME=" + user); //
Sensitive

ResultSet rs3 = pstmt.executeQuery();
```

Risoluzione

```
Statement stmt1 = con.createStatement();

ResultSet rs1 = stmt1.executeQuery("select FNAME, LNAME,
SSN” + “from USERS"); // No issue; hardcoded query

String query = "select FNAME, LNAME, SSN from USERS where
UNAME=?"

PreparedStatement pstmt = con.prepareStatement(query);

pstmt.setString(1, user); // Good; PreparedStatements escape
their inputs.

ResultSet rs2 = pstmt.executeQuery();
```

DISABLING RESOURCE INTEGRITY FEATURES

L'acquisizione di **risorse esterne** senza verificarne l'integrità può compromettere la sicurezza di un'applicazione se la fonte da cui proviene viene compromessa. È possibile verificare l'integrità di una risorsa esterna nel seguente modo:

1. Nel codice sorgente, la risorsa viene "etichettata" con una stringa alfanumerica (*digest*) generata dalla codifica del suo contenuto;
2. Quando un'applicazione accede alla risorsa esterna, viene calcolato il *digest* sulla base del contenuto della risorsa a cui si sta tentando di accedere;
3. Se il *digest* con cui la risorsa è stata "etichettata" è uguale a quello calcolato al momento in cui si sta tentando l'accesso, allora significa che la risorsa che si sta recuperando è quella che ci si aspetta (integra), altrimenti è malevola.

Security Hotspot

```
<script src="https://cdnexample.com/script.js"></script>

<!-- Sensitive -->
```

Risoluzione

```
<script src="https://cdnexample.com/script.js"
integrity="sha384-
oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4Jw
Y8wC"></script>

<!-- Compliant: integrity value should be replaced with the
digest of the expected resource -->
```

USING SLOW REGULAR EXPRESSIONS

Le espressioni regolari permettono di **controllare la validità degli input** inseriti dagli utenti. Quando la stringa in input risulta essere molto lunga o l'espressione regolare molto complessa, la computazione può richiedere tantissimo tempo: inviando molteplici richieste HTTP in breve tempo, un attaccante potrebbe causare un denial of service dell'applicazione.

Per prevenire tali situazioni, è consigliabile utilizzare espressioni regolari minimali (perlomeno senza inutili ripetizioni), definire una lunghezza massima per l'input fornito dall'utente o impostare dei time-out oltre i quali la computazione termina.

Security Hotspot

```
import re


pattern = "(a+)+"

# Questo pattern causerà un blocco infinito quando
utilizzato con una stringa lunga contenente molti caratteri
"a" consecutivi

re.match(pattern, "a" * 1000000)
```

Risoluzione

```
import re


pattern = "a+"

# Questo pattern non causerà problemi di prestazioni con una
stringa lunga contenente molti caratteri "a" consecutivi

re.match(pattern, "a" * 1000000)
```

HARD-CODED PASSWORDS

Poiché è facile **estrarre stringhe** dal codice sorgente o dal codice binario di un'applicazione, le password non dovrebbero essere codificate in modo rigido, cioè direttamente nel codice. Tale problematica risulta particolarmente rilevante per le applicazioni distribuite o open source.

È consigliabile che le password siano memorizzate al di fuori del codice in un file di configurazione, in un database o in un servizio di gestione delle password.

Security Hotspot

```
String username = "steve";

String password = "blue";

Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost/test?" +
"user=" + uname + "&password=" + password);

// Sensitive
```

Risoluzione

```
String username = getEncryptedUser();

String password = getEncryptedPassword();

Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost/test?" +
"user=" + uname + "&password=" + password);
```

USING PSEUDORANDOM NUMBER GENERATORS

I generatori di numeri pseudo-casuali (PRNG, dall'inglese pseudo-random number generator) sono algoritmi **deterministici** che permettono di creare sequenze numeriche **prevedibili**; utilizzarli in contesti in cui è richiesta **imprevedibilità** può costituire una minaccia per il sistema perché un attaccante potrebbe indovinare e sfruttare i numeri generati, per questo motivo è bene preferire l'utilizzo di generatori di numeri casuali (RNG, dall'inglese number generator) crittograficamente forti.

Security Hotspot

```
Random random = new Random();
// Sensitive use of Random
byte bytes[] = new byte[20];
random.nextBytes(bytes);
// Check if bytes is used for hashing, encryption, etc...
```

Risoluzione

```
SecureRandom random = new SecureRandom();
// Compliant for security-sensitive use cases
byte bytes[] = new byte[20];
random.nextBytes(bytes);
```

AUTHORIZING AN OPENED WINDOW TO ACCESS BACK TO THE ORIGINATING WINDOW

Il tag <a href=> viene utilizzato per collegare una pagina web X ad una pagina web Y. Utilizzando l'attributo target="_blank", quando l'utente clicca sul collegamento ipertestuale nella pagina web X, la pagina web Y verrà aperta in un nuova scheda del browser (quindi la pagina web X rimarrà ancora **aperta**).

Tale comportamento può costituire una minaccia alla sicurezza del sistema perché la pagina web Y potrebbe **accedere** e modificare la pagina web X, ad esempio cambiandola in una pagina web Z malevola che chiede all'utente l'inserimento di dati sensibili. Per evitare ciò, è possibile utilizzare l'attributo rel=noopener.

Security Hotspot

```
<a href="http://example.com/dangerous" target="_blank">
<!-- Sensitive -->

<a href="{{variable}}" target="_blank">
<!-- Sensitive -->
```

Risoluzione

```
<a href="http://petssocialnetwork.io" target="_blank"
rel="noopener"> <!-- Compliant -->
```

USING CLEAR-TEXT PROTOCOLS

L'utilizzo di protocolli per lo scambio di informazioni che non ne prevedono la crittografia durante il **trasporto** può permettere a malintenzionati di leggerne e modificarne il contenuto, in quanto il traffico di rete può essere **intercettato**. È possibile evitare tali pericoli impiegando protocolli sicuri:

- telnet può essere sostituito con ssh ;
- ftp può essere sostituito con sftp, scp, ftps;
- http può essere sostituito con https.

Security Hotspot

```
ConnectionSpec spec = new
ConnectionSpec.Builder(ConnectionSpec.CLEARTEXT).build();

// Sensitive
```

Risoluzione

```
ConnectionSpec spec = new
ConnectionSpec.Builder(ConnectionSpec.MODERN_TLS).build();

// Compliant
```

DYNAMICALLY EXECUTING CODE

L'esecuzione del codice in modo **dinamico** (ad esempio sulla base degli input forniti da utenti) può costituire un pericolo di injection sia lato client che lato server.

È consigliato non eseguire mai codice sconosciuto proveniente da fonti non attendibili

Security Hotspot

```
var greeting = "good morning"
function speak(str) {
  eval(str) // Sensitive
  console.log(greeting)
}
speak("var greeting = 'meow'")
```

Risoluzione

```
var morning = "good morning"
function speak(greeting) {
  console.log(morning)
}
speak(morning)
// Compliant
```

ENCRYPTION ALGORITHMS SHOULD BE USED WITH SECURE MODE AND PADDING SCHEME

Gli algoritmi di crittografia dovrebbero utilizzare **modalità sicure** e **schemi di padding** dove appropriati per garantire la confidenzialità e l'integrità dei dati.

Per gli algoritmi di crittografia a blocchi (come AES), bisognerebbe prevedere l'utilizzo della modalità GCM al posto di ECB e CBC; invece, per l'algoritmo di crittografia RSA è indicato lo schema di padding OAEP.

Vulnerability

```
Cipher.getInstance("AES"); // Noncompliant: by default ECB mode is chosen

Cipher.getInstance("AES/ECB/NoPadding"); // Noncompliant: ECB doesn't provide serious message confidentiality

Cipher.getInstance("AES/CBC/PKCS5Padding"); // Noncompliant: Vulnerable to Padding Oracle attacks

Cipher.getInstance("RSA/None/NoPadding"); // Noncompliant: RSA without OAEP padding scheme is not recommended
```

Risoluzione

```
Cipher.getInstance("AES/GCM/NoPadding");

Cipher.getInstance("RSA/None/OAEPWITHSHA-256ANDMGF1PADDING"); // or the ECB mode can be used for RSA when "None" is not available with the security provider used - in that case, ECB will be treated as "None" for RSA.

Cipher.getInstance("RSA/ECB/OAEPWITHSHA-256ANDMGF1PADDING");
```

CIPHER ALGORITHMS SHOULD BE ROBUST

Gli algoritmi di cifratura forti sono resistenti alla crittoanalisi e alle più note tipologie di attacchi, come quelli a forza bruta.

Una raccomandazione generale è quella di utilizzare solo algoritmi di cifratura **intensamente testati e promossi dalla comunità crittografica**.

Vulnerability

```
Cipher c1 = Cipher.getInstance("DES");

// Noncompliant: DES works with 56-bit keys allow attacks
via exhaustive search
```

Risoluzione

```
Cipher c31 = Cipher.getInstance("AES/GCM/NoPadding");

// Compliant
```

USING WEAK HASHING ALGORITHMS

Algoritmi di **hash crittografici** (es. MD5, SHA-1) non sono più considerati sicuri, perché basta un piccolo sforzo computazionale per trovare due o più input diversi che producono lo stesso hash. Quando è necessario garantire elevati standard di sicurezza, è consigliato utilizzare algoritmi più sicuri per effettuare l'hashing (es. SHA-256, SHA-512).

Security Hotspot

```
MessageDigest md1 = MessageDigest.getInstance("SHA");

// Sensitive:  SHA is not a standard name, for most security providers it's an alias of SHA-1


MessageDigest md2 = MessageDigest.getInstance("SHA1");

// Sensitive
```

Risoluzione

```
MessageDigest md1 = MessageDigest.getInstance("SHA-512");

// Compliant
```