LIPSA SUPRASCRIERII MEMORIEI DUPĂ OPERAȚII CU DATE SENSIBILE (CITIREA DE LA CONSOLĂ)

Datele sensibile din memorie pot fi compromise. Un adversar care poate executa cod pe același sistem cu o altă aplicație poate accesa datele sensibile dacă aceasta utilizează obiecte pentru a le stoca, iar conținutul nu este șters după utilizare, are pagini de memorie care pot fi trimise pe disc după cum este cerut de sistemul de operare (de exemplu, pentru a efectua sarcini de gestionare a memoriei sau pentru a hibern), ține date sensibile într-un buffer intern (cum ar fi cel folosit de BufferedReader) care păstrează copii ale datelor în memorie sau le dezvăluie în mesajele de depanare sau fișierele jurnal.

Scurgerile de date sensibile devin mai probabile dacă memoria care conține datele nu este suprascrisă după utilizarea acestora. Pentru a limita riscul de expunere, programele trebuie să reducă la minimum durata de viață a datelor sensibile.

Atenuarea completă necesită suport din partea sistemului de operare și a mașinii virtuale Java. Dacă trimiterea datelor sensibile pe disc constituie o problemă, este necesar un sistem de operare sigur care să dezactiveze mecanismul de swapping și hibernarea.

Acest exemplu de cod neconform citește informațiile despre numele de utilizator și parola din consolă și stochează parola ca obiect de tip String. Credențialele rămân expuse până când garbage collector-ul recuperează memoria asociată cu acest String.

```
4   class Password {
        Run | Debug
5       public static void main(String args[]) throws IOException {
6           Console c = System.console();
7           if (c == null) {
8               System.err.println(x:"No console.");
9               System.exit(status:1);
10          }
11          String username = c.readLine(fmt:"Enter your username: ");
12          String password = c.readLine(fmt:"Enter your password: ");
13          if (!verify(username, password)) {
14              throw new SecurityException(s:"Invalid Credentials");
15          }
16          // ...
17      }
18
19      // dummy verify method, always returns true
20      private static final boolean verify(String username, String password) {
21          return true;
22      }
23  }
```

```
PS C:\workspace\secure-coding-practices\Java\UncleanedCredentialsAfter>  c:; cd 'c:\workspa
ce\secure-coding-practices\Java\UncleanedCredentialsBefore'; & 'C:\Program Files\Java\jdk-2
0\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\dpili\AppData\Roa
ming\Code\User\workspaceStorage\ffd95426091feff0354ed86ae0bc5053\redhat.java\jdt_ws\Unclean
edCredentialsBefore_547df7c3\bin' 'Password'
Enter your username: tomas
Enter your password: byshel
```

Această soluție conformă utilizează metoda Console.readPassword() pentru a obține parola de la consolă. Metoda Console.readPassword() permite returnarea parolei ca un șir de caractere, decât ca un obiect String. În consecință, programatorul poate șterge parola din șir imediat după utilizare. Această metodă dezactivează, de asemenea, afișarea parolei pe consolă.

```
You, 4 days ago | 1 author (You)
class Password {
    Run | Debug
    public static void main(String args[]) throws IOException {
        Console c = System.console();
        if (c == null) {
            System.err.println(x:"No console.");
            System.exit(status:1);
        }
        String username = c.readLine(fmt:"Enter your user name: ");
        char[] password = c.readPassword(fmt:"Enter your password: ");
        if (!verify(username, password)) {
            throw new SecurityException(s:"Invalid Credentials");
        }
        // clear the password
        Arrays.fill(password, val:' ');
    }

    // dummy verify method, always returns true
    private static final boolean verify(String username, char[] password) {
        return true;
    }
}
```

```
PS C:\workspace\secure-coding-practices\Java\UncleanedCredentialsAfter>  c:; cd 'c:\workspa
ce\secure-coding-practices\Java\UncleanedCredentialsAfter'; & 'C:\Program Files\Java\jdk-20
\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\dpili\AppData\Roam
ing\Code\User\workspaceStorage\ffd95426091feff0354ed86ae0bc5053\redhat.java\jdt_ws\Uncleane
dCredentialsAfter_76493518\bin' 'Password'
Enter your user name: tomas
Enter your password:
```

În Java, odată ce un obiect String este creat, acesta este imutabil, ceea ce înseamnă că nu îi poate fi schimbat conținutul. De aceea, un obiect String cu conținut sensibil nu poate fi suprascris. În schimb, poate fi creat un nou String cu conținut gol și atribuit aceleiași variabile, suprascriind efectiv referința la String-ul original. Cu toate acestea, ștergerea unui String în acest fel nu elimină datele sensibile din memorie imediat. Obiectul String original există încă în memorie până când este colectat de garbage collector. Acest

lucru poate reprezenta un risc de securitate dacă memoria este accesată de un atacator și unul dintre motivele pentru care este recomandată utilizarea șirurilor de caractere pentru a stoca informații sensibile. Conținutul șirurilor de caractere poate fi golit în mod explicit prin suprascrierea cu o valoare cunoscută (de exemplu, spații) imediat după utilizare. Acest lucru oferă mai mult control asupra datelor sensibile și reduce riscul expunerii în memorie.

LIPSA SUPRASCRIERII MEMORIEI DUPĂ OPERAȚII CU DATE SENSIBILE (CITIREA DIN FIȘIER)

Acest exemplu de cod neconform utilizează un BufferedReader pentru a încadra un obiect InputStreamReader, astfel încât datele sensibile să poată fi citite dintr-un fișier. Metoda BufferedReader.readLine() returnează datele sensibile ca obiect String, care poate persista mult timp după ce datele nu mai sunt necesare.

```
You, 1 second ago | 1 author (You)
public class File {
    Run | Debug
    public static void main(String args[]) throws IOException {
        String filePath = System.getProperty(key:"user.dir") + "\\src\\secret.txt'
        // try-with-resources block automatically closes the BufferedReader
        try (BufferedReader br = new BufferedReader(new InputStreamReader(
                new FileInputStream(filePath)))) {
            String data;
            while ((data = br.readLine()) != null)
                System.out.println(data);
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

```
PS C:\workspace\secure-coding-practices\Java\UncleanedFileContentBefore>  & 'C:\Program Fil
es\Java\jdk-20\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\dpil
i\AppData\Roaming\Code\User\workspaceStorage\ffd95426091feff0354ed86ae0bc5053\redhat.java\j
dt_ws\UncleanedFileContentBefore_61093b64\bin' 'File'
SSBzb2xlbW5seSBzd2VhciB0aGF0IEkgYW0gdXAgdG8gbm8gZ29vZC4uLg==
```

Această soluție conformă utilizează un buffer NIO (new I/O) alocat direct pentru a citi datele sensibile din fișier. Datele pot fi șterse imediat după utilizare și nu sunt memorate în cache sau în buffere interne din mai multe locații, existând numai în memoria sistemului. Ștergerea manuală a datelor din buffer este obligatorie deoarece buffere-le directe nu sunt colectate de garbage collector.

```
You, 4 days ago | 1 author (You)
class File {
    Run | Debug
    public static void main(String args[]) throws IOException {

        String filePath = System.getProperty(key:"user.dir") + "\\src\\secret.txt'
        ByteBuffer buffer = ByteBuffer.allocateDirect(capacity:30);
        byte[] zeroes = new byte[buffer.capacity()];

        try (FileChannel channel = (new FileInputStream(filePath)).getChannel()) {
            while (channel.read(buffer) != -1) {
                buffer.flip();
                while (buffer.hasRemaining()) {
                    System.out.print((char) buffer.get());
                }
                buffer.clear();
                buffer.put(zeroes);
                buffer.clear();
            }

        } catch (Throwable e) {
            e.printStackTrace();
        }

    }
}
```

```
PS C:\workspace\secure-coding-practices\Java\UncleanedFileContentBefore>  c:; cd 'c:\worksp
ace\secure-coding-practices\Java\UncleanedFileContentAfter'; & 'C:\Program Files\Java\jdk-2
0\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\dpili\AppData\Roa
ming\Code\User\workspaceStorage\ffd95426091feff0354ed86ae0bc5053\redhat.java\jdt_ws\Unclean
edFileContentAfter_4d667957\bin' 'File'
SSBzb2xlbW5seSBzd2VhciB0aGF0IEkgYW0gdXAgdG8gbm8gZ29vZC4uLg==
```

ByteBuffer face parte din framework-ul Java NIO, care oferă o modalitate mai eficientă pentru citirea sau scrierea unor cantități mari de date și este proiectat să funcționeze cu sistemul de gestionare a memoriei Java. Permite alocarea și lucrul cu un buffer direct, care se află într-o regiune de memorie în afara heap-ului. În soluția conformă, un ByteBuffer este utilizat pentru a citi datele sensibile din fișier. FileChannel oferă metode pentru citirea datelor direct într-un ByteBuffer, care este o modalitate eficientă de a transfera date între fișier și memorie. BufferedReader citește datele într-un String și poate fi mai puțin eficient din punct de vedere al memoriei, în special pentru fișierele mari.

Când datele sunt citite într-un ByteBuffer, poziția buffer-ului este avansată până la sfârșitul datelor care au fost citite și limita este setată la aceeași valoare ca poziția. Acest lucru reflectă faptul că buffer-ul conține date valide până la poziția curentă. Pentru a reutiliza buffer-ul pentru citirea următorului calup de date, trebuie resetată atât poziția, cât și limita la valorile inițiale apelând buffer.clear().

După procesarea datelor din calupul curent, bucla continuă să citească următorul calup de date în același mod, iar buffer-ul este șters din nou pentru a se pregăti pentru următoarea operație de citire.

ABSENȚA UNOR CLASE WRAPPER CARE SĂ ASIGURE IMUTABILITATEA DATELOR SENSIBILE

Acest exemplu de cod neconform constă din clasa User, care permite modificarea șirului de caractere intern. Un invocator care nu este de încredere poate apela metoda setBankAccountNumber() și poate încălca proprietatea de imutabilitate a obiectului, permițând codului extern să înlocuiască șirul de caractere cu un alt șir. Invocarea metodei getBankAccountNumber() permite, de asemenea, modificarea nerestricționată a stării interne private a clasei, prin returnarea unei referințe la membrul mutabil bankAccountNumber. Exemplul de cod neconform furnizat încalcă principiile de securitate și încapsulare prin expunerea accesului direct la membrul mutabil privat și permițând codului extern să îl modifice.

```
You, 4 days ago | 1 author (You)
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        SubmittedUserForm userData = new SubmittedUserForm();
        UserService.updateUser(userData);
        UserService.displayUser(userData.userId);
    }
}
```

```
    public static void displayUser(int userId) {
        // pretend to retrieve data from the db based on userId
        User user = new User();
        renderUser(user);
    }

    public static void renderUser(User user) {
        // programmer is allowed to change data even though display doesn't imply
        user.setBankAccountNumber("BCR1234567887654321".toCharArray());
        System.out.println(user.getBankAccountNumber());
    }
```

```
public class User {
    private int userId = 1;
    // other important fields like firstname = Maricica, lastname = Morrone
    // data set a long time ago when the user registered
    private char[] bankAccountNumber = "ING1234567812345678".toCharArray();

    // getters and setters for the other fields

    public char[] getBankAccountNumber() {
        return bankAccountNumber;
    }

    public void setBankAccountNumber(char[] bankAccountNumber) {
        this.bankAccountNumber = bankAccountNumber;
    }
}
```

```
PS C:\workspace\secure-coding-practices\Java\SensitiveMutableClassAfter>  c:; cd 'c:\worksp
ace\secure-coding-practices\Java\SensitiveMutableClassBefore'; & 'C:\Program Files\Java\jdk
-20\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\dpili\AppData\R
oaming\Code\User\workspaceStorage\ffd95426091feff0354ed86ae0bc5053\redhat.java\jdt_ws\Sensi
tiveMutableClassBefore_1d6561d0\bin' 'App'
ING9876543298765432
BCR1234567887654321
```

În general, clasele sensibile pot fi transformate în obiecte sigur de vizualizat prin furnizarea de wrappere adecvate pentru toate metodele definite de interfața de bază. Wrapper-ele trebuie să arunce o excepție UnsupportedOperation, astfel încât clienții să nu poată efectua operații care afectează proprietatea de imutabilitate a obiectului.

Clasa wrapper ImmutableUser suprascrie metoda getBankAccountNumber() și clonează șirul de caractere. Apelantul primește o copie a șirului, iar șirul original rămâne neschimbat și inaccesibil. Metoda setBankAccountNumber() aruncă o excepție dacă apelantul încearcă să utilizeze această metodă pe obiectul returnat. Acest obiect poate fi transmis unui cod care nu este de încredere atunci când este permisă citirea datelor.

```
11      public static void displayUser(int userId) {
12          // pretend to retrieve data from the db based on userId
13          ImmutableUser user = new ImmutableUser(new User());
14          renderUser(user);
15      }
16
17      public static void renderUser(ImmutableUser user) {
18          // programmer is no longer allowed to change data
19          user.setBankAccountNumber("BCR1234567887654321".toCharArray());
20          System.out.println(user.getBankAccountNumber());
21      }
```

```
1   public class ImmutableUser extends User {
2       ImmutableUser(User user) {
3           super(user);
4       }
5
6       @Override
7       public char[] getBankAccountNumber() {
8           return super.getBankAccountNumber().clone();
9       }
10
11      @Override
12      public void setBankAccountNumber(char[] bankAccountNumber) {
13          throw new UnsupportedOperationException();
14      }
15  }
```

```
PS C:\workspace\secure-coding-practices\Java\SensitiveMutableClassAfter>  & 'C:\Program Fil
es\Java\jdk-20\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\dpil
i\AppData\Roaming\Code\User\workspaceStorage\ffd95426091feff0354ed86ae0bc5053\redhat.java\j
dt_ws\SensitiveMutableClassAfter_42f5d56b\bin' 'App'
ING9876543298765432
java.lang.UnsupportedOperationException
        at ImmutableUser.setBankAccountNumber(ImmutableUser.java:13)
        at UserService.renderUser(UserService.java:19)
        at UserService.displayUser(UserService.java:14)
        at App.main(App.java:6)
```

Imutabilitatea membrilor previne modificarea accidentală, precum și manipularea rău intenționată, iar copierea defensivă (clonarea) nu mai este necesară, însă unele clase sensibile nu pot fi imutabile. Astfel, accesul numai în modul citire la clasele mutabile poate fi acordat codului care nu este de încredere folosind wrappere nemodificabile. Un exemplu sunt clasele Collection care includ un set de wrappere ce permit clienților să observe un obiect Collection nemodificabil. Nefurnizarea unei versiuni nemodificabile a unui obiect sensibil mutabil unui cod care nu este de încredere poate duce la manipularea și coruperea rău intenționată a obiectului.

O clasă mutabilă sensibilă poate fi o clasă care reprezintă conturi de utilizator sau profiluri dintr-un sistem, cum ar fi parole, adrese de e-mail sau detalii personale, o clasă care reprezintă tranzacții financiare, cum ar fi tranzacțiile cu cardul de credit care ar putea conține informații sensibile, cum ar fi sumele tranzacțiilor, numerele de cont și marcajele temporale, o clasă care conține înregistrări medicale pentru pacienți care ar putea conține date sensibile, cum ar fi istoricul medical, diagnosticele și planurile de tratament sau o clasă responsabilă pentru generarea, stocarea și gestionarea cheilor criptografice care sunt extrem de sensibile și trebuie protejate împotriva accesului neautorizat.

VULNERABILITĂȚI ÎN GESTIONAREA FIȘIERELOR

ABSENȚA VERIFICĂRII ȘI VALIDĂRII CĂILOR CĂTRE FIȘIERE + ATAC DIRECTORY PATH TRAVERSAL

Un nume de cale poate fi absolut sau relativ. Un nume de cale absolută este complet prin faptul că nu sunt necesare alte informații pentru a localiza fișierul pe care îl denotă. În schimb, un nume de cale relativă trebuie interpretat în funcție de informațiile preluate dintr-un alt nume de cale. Numele absolute sau relative ale căilor pot conține legături simbolice de fișiere. Aceste legături de fișiere trebuie rezolvate complet înainte de efectuarea oricăror operațiuni de validare a fișierelor. Numele căilor pot conține, de asemenea, nume speciale de fișiere care îngreunează validarea, precum "." care se referă la directorul în sine sau ".." care se referă la directorul părinte.

Canonizarea numelor de fișiere facilitează validarea unui nume de cale. Mai multe nume de cale se pot referi la un singur director sau fișier. Mai mult, reprezentarea textuală a unui nume de cale poate oferi puține informații sau deloc cu privire la directorul sau fișierul la care se referă. În consecință, toate numele căilor trebuie să fie complet rezolvate sau canonizate înainte de validare.

Validarea este necesară atunci când se încearcă restricționarea accesului utilizatorului la fișierele dintr-un anumit director sau când sunt luate decizii de securitate pe baza unui nume de fișier sau a numelui căii. Frecvent, aceste restricții pot fi eludate de un atacator prin exploatarea unei vulnerabilități de traversare a directorului sau de echivalență a căii. O vulnerabilitate de traversare a directorului permite unei operațiuni I/O să nu opereze într-un director specificat. O vulnerabilitate de echivalență a căii apare atunci când un atacator furnizează un nume diferit, dar echivalent pentru o resursă pentru a ocoli verificările de securitate.

Canonizarea conține o fereastră de timp facilitatoare pentru o condiție de cursă între momentul în care programul obține numele căii canonizate și momentul în care deschide fișierul. În timp ce numele căii canonice este validat, este posibil ca sistemul de fișiere să fi fost modificat și numele căii canonice să nu mai facă referire la fișierul valid original. Această condiție de cursă poate fi ușor mitigată dacă fișierul la care se face referire se află într-un director securizat. Atunci, prin definiție, un atacator nu îl poate manipula și nu poate exploata condiția de cursă. În consecință, nu sunt recomandate operațiile pe fișierele din directoarele partajate.

Acest exemplu de cod neconform permite utilizatorului să specifice calea unui fișier de deschis. Prin alipirea la numele fișierului a directorului de lucru curent și a directorului uploads, acest cod impune o politică conform căreia numai fișierele din acest director ar trebui deschise. Cu toate acestea, utilizatorul poate specifica în continuare un fișier în afara directorului intenționat introducând un argument care conține secvențe de tipul “..”. Un atacator poate crea, de asemenea, un o legătură simbolică în directorul uploads care se referă la un director sau fișier din afara acelui director. Numele căii legăturii simbolice ar putea părea că se află în directorul uploads și, în consecință, operația va fi efectuată de fapt pe ținta finală a acesteia.

```java
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {

        BufferedReader reader = new BufferedReader(
                new InputStreamReader(System.in));
        System.out.print(s:"Enter a file name: ");
        String userInput = reader.readLine();

        Path baseDir = Paths.get(System.getProperty(key:"user.dir"))
                .resolve(other:"uploads");
        Path filePath = baseDir.resolve(userInput)

        byte[] fileContent = Files.readAllBytes(filePath);
        if (fileContent != null) {
            System.out.print(s:"File content: ");
            for (int i = 0; i < fileContent.length; i++) {
                System.out.print((char) fileContent[i]);
                fileContent[i] = 0;
            }
        } else {
            System.out.println(x:"Failed to read file.");
        }

    }
}
```

```
denissa@denissa-vm:~/Desktop/secure-coding-practices/Java/CanonicalizePathNameBefore$  /usr/bin/env
 /usr/lib/jvm/jdk-21-oracle-x64/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -
cp /home/denissa/.config/Code/User/workspaceStorage/b2005b83808ca836c42320944af10df1/redhat.java/jd
t_ws/CanonicalizePathNameBefore_d2ddd5a/bin App
Enter a file name: ../../../../../../../etc/passwd
File content: root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

În exemplul de cod conform, canonizarea căii fișierului s-a realizat prin apelul funcțiilor normalize() și toAbsolutePath() pe obiectul aferent căii fișierului. Astfel, sunt rezolvate legăturile simbolice, dacă există, și sunt eliminate orice componente de cale potențial periculoase precum “..”, pentru a se asigura faptul că este indicată o locație sigură. Verificarea căii canonizate cu calea directorului de bază poate oferi un nivel rezonabil de securitate împotriva atacurilor de traversare a directoarelor, asigurând operațiile cu fișiere în cadrul directorului uploads.

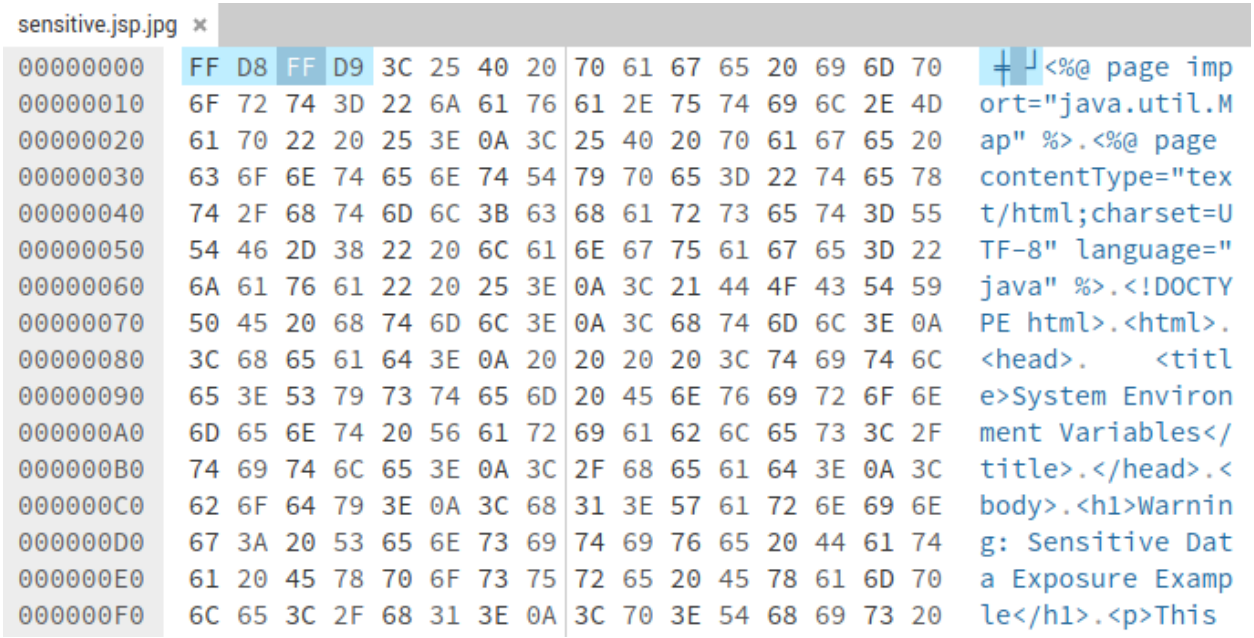
```java
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {

        BufferedReader reader = new BufferedReader(
                new InputStreamReader(System.in));
        System.out.print(s:"Enter a file name: ");
        String userInput = reader.readLine();

        Path baseDir = Paths.get(System.getProperty(key:"user.dir"))
                .resolve(other:"uploads");
        Path filePath = baseDir.resolve(userInput).normalize().toAbsolutePath()

        if (!isValidFilePath(filePath, baseDir)) {
            System.out.println(x:"Invalid file path.");
            return;
        }

        byte[] fileContent = Files.readAllBytes(filePath);
        if (fileContent != null) {
            System.out.print(s:"File content: ");
            for (int i = 0; i < fileContent.length; i++) {
                System.out.print((char) fileContent[i]);
                fileContent[i] = 0;
            }
        } else {
            System.out.println(x:"Failed to read file.");
        }

    }

    public static boolean isValidFilePath(Path filePath, Path baseDir) {
        // ensure that filePath doesn't contain directory traversal attempts
        // and that it is within the specified baseDir
        return filePath.startsWith(baseDir) &&
            !filePath.toString().contains(s:"..");
    }
}
```

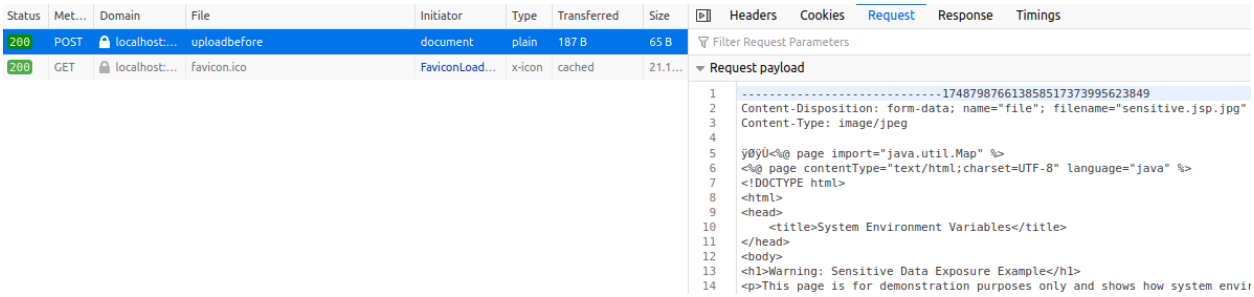
```
denissa@denissa-vm:~/Desktop/secure-coding-practices/Java/CanonicalizePathNameAfter$
  /usr/bin/env /usr/lib/jvm/jdk-21-oracle-x64/bin/java --enable-preview -XX:+ShowCod
eDetailsInExceptionMessages -cp /home/denissa/.config/Code/User/workspaceStorage/b20
05b83808ca836c42320944af10df1/redhat.java/jdt_ws/CanonicalizePathNameAfter_adcac0a1/
bin App
Enter a file name: ../../../../../../../etc/passwd
Invalid file path.
```

ABSENȚA UNEI VERIFICĂRI ROBUSTE A TIPULUI DE FIȘIER + ATAC ARBITRARY FILE UPLOAD

Încărcarea arbitrară a fișierelor sau încărcarea nerestricționată de fișiere este o vulnerabilitate de securitate care apare în aplicațiile web atunci când un atacator este capabil să încarce și să execute sau să stocheze fișiere arbitrare pe server. Acest lucru poate duce la diverse riscuri de securitate și atacuri dacă nu este mitigat corespunzător. Atacatorii pot încărca fișiere infectate cu malware (viruși, troieni) pe server, compromițând securitatea și integritatea serverului. Atacatorii pot încărca, de asemenea, scripturi webshell, care sunt scripturi rău intenționate ce le pot permite să obțină acces neautorizat la server, să execute cod arbitrar și să efectueze diverse acțiuni rău intenționate.

În acest exemplu, o pagină Java Server Page mascată ca imagine JPEG este încărcată pentru a obține informații sensibile despre variabilele sistemului de mediu. Nu numai extensia este modificată, ci și numărul magic corespunzător formatului de imagine JPEG prezent la începutul fișierului.





O aplicație care verifică doar extensia și câmpul Content-Type din antetul HTTP este vulnerabilă la un astfel de atac. Încărcarea fișierului trebuie să reușească numai atunci când tipul de conținut se potrivește cu conținutul real al fișierului. Un fișier cu antet de imagine trebuie să conțină doar o imagine și nu trebuie să conțină cod executabil.

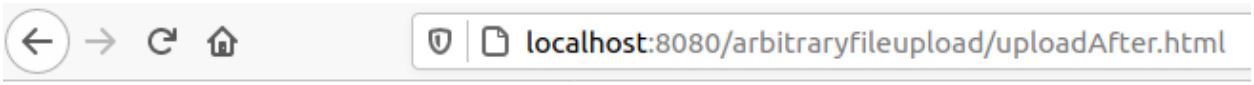
```
25      private final List<String> acceptedContentTypes = Arrays.asList(
26              ...a:"image/jpeg",
27              "image/png");
```

```
private void uploadBefore(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    Part filePart = request.getPart(name:"file");
    Path filePath = uploadPath.resolve(filePart.getSubmittedFileName());
    String mimeTypeFromContentType = filePart.getContentType();
    String mimeFromExtension = Files.probeContentType(filePath);

    if (filePart != null && filePart.getSize() > 0) {
        if (acceptedContentTypes.contains(mimeTypeFromContentType) &&
                acceptedContentTypes.contains(mimeFromExtension) &&
                mimeTypeFromContentType.equals(mimeFromExtension)) {
            filePart.write(filePath.toString());
            response.getWriter().println("File uploaded successfully to " + filePath);
        } else {
            response.getWriter().println(x:"The file type is not supported or does not
        }
    } else {
        response.getWriter().println(x:"No file uploaded.");
    }
}
```

Această soluție conformă utilizează biblioteca Apache Tika pentru a detecta și extrage metadate din documente utilizând bibliotecile de parsatoare existente. Este o alegere potrivită pentru detectarea precisă a tipului de conținut pentru o gamă largă de formate de fișiere și extragerea detaliată a metadatelor. Parser-ul Apache Tika analizează întregul conținut al fișierului și extrage metadatele, nu verifică doar tipul MIME al fișierului pe baza unei examinări rapide a octeților de început ai conținutului fișierului. Tipul de conținut și extensia fișierului încărcat, precum și potrivirea acestora sunt verificate înainte de a face analiza aprofundată a conținutului. Acest proces în mai mulți pași evită analizarea completă a fișierelor care nu se potrivesc cu unul dintre tipurile MIME acceptate, economisind astfel resurse de sistem și timp de procesare. Metadatele specifice pentru tipul de fișier încărcat trebuie verificate înainte de a scrie fișierul pe disc.

```
88          if (acceptedContentTypes.contains(mimeTypeFromContentType) &&
89                  acceptedContentTypes.contains(mimeFromExtension) &&
90                  mimeTypeFromContentType.equals(mimeFromExtension) &&
91                  checkMagicAndMetadata(filePart)) {
92              filePart.write(filePath.toString());
93              response.getWriter().println("File uploaded successfully to " + filePath);
94          } else {
95              response.getWriter().println(x:"The file type is not supported or does not
96          }
```

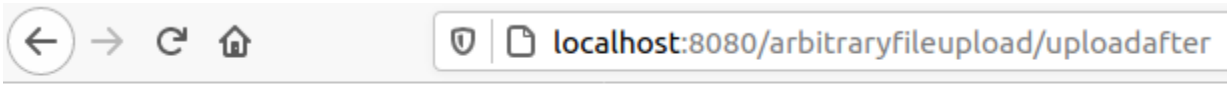
```
102     private boolean checkMagicAndMetadata(Part filePart) {
103         AutoDetectParser parser = new AutoDetectParser();
104         BodyContentHandler handler = new BodyContentHandler();
105         Metadata metadata = new Metadata();
106         try (InputStream is = TikaInputStream.get(filePart.getInputStream())) {
107             parser.parse(is, handler, metadata);
108             boolean typeIsCorrect = metadata.get(Metadata.CONTENT_TYPE)
109                     .equalsIgnoreCase(filePart.getContentType());
110             boolean dimensionsExist = (metadata.get(Metadata.IMAGE_WIDTH) != null &&
111                     metadata.get(Metadata.IMAGE_LENGTH) != null);
112             return typeIsCorrect && dimensionsExist;
113         } catch (IOException | SAXException | TikaException e) {
114             return false;
115         }
116     }
```



Upload a File

Select a file: Browse... sensitive.jsp.jpg

Upload




```
The file type is not supported or does not match the expected content.
```

Pentru a preveni vulnerabilitățile de încărcare arbitrară a fișierelor, dezvoltatorii ar trebui să implementeze măsuri de securitate adecvate, cum ar fi validarea tipurilor de fișiere și a extensiilor, pentru a se asigura că sunt permise numai tipurile de fișiere sigure și redenumirea fișierelor încărcate pentru a evita potențialele conflicte sau riscurile de execuție.

SUPRASCRIEREA UNUI FIȘIER ÎN MOMENTUL CREĂRII UNUI FIȘIER CU ACELAȘI NUME

Acest exemplu de cod neconform încearcă să deschidă un fișier pentru scriere. Dacă fișierul a existat înainte de a fi deschis, conținutul său anterior va fi suprascris cu conținutul furnizat de program. Existența unui fișier trebuie verificată. Dacă există deja un fișier cu numele dat, atunci crearea nu ar trebui să se întâmple.

```
6   public class App {
        Run | Debug
7       public static void main(String[] args) throws Exception {
8           String filename = "example.txt";
9           String directoryPath = "/var/www/uploads";
10          Path filePath = Paths.get(directoryPath, filename);
11          String content = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut ur
12
13          try {
14              Files.createDirectories(filePath.getParent());
15              Files.write(filePath, content.getBytes());
16              System.out.println("File created successfully with content at " + filePath);
17          } catch (IOException e) {
18              System.err.println("Error occurred while writing to the file: " + e.getMessa
19          }
20      }
21  }
```

```
denissa@ubuntu:~/Desktop/secure-coding-practices/Java/FileCreationOverwriteBefore$  cd /home/de
nissa/Desktop/secure-coding-practices/Java/FileCreationOverwriteBefore ; /usr/bin/env /usr/lib/
jvm/jdk-21-oracle-x64/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /ho
me/denissa/.config/Code/User/workspaceStorage/b2005b83808ca836c42320944af10df1/redhat.java/jdt_
ws/FileCreationOverwriteBefore_b7bf8e4/bin App
File created successfully with content at /var/www/uploads/example.txt
denissa@ubuntu:~/Desktop/secure-coding-practices/Java/FileCreationOverwriteBefore$  cd /home/de
nissa/Desktop/secure-coding-practices/Java/FileCreationOverwriteBefore ; /usr/bin/env /usr/lib/
jvm/jdk-21-oracle-x64/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /ho
me/denissa/.config/Code/User/workspaceStorage/b2005b83808ca836c42320944af10df1/redhat.java/jdt_
ws/FileCreationOverwriteBefore_b7bf8e4/bin App
File created successfully with content at /var/www/uploads/example.txt
```

Java oferă mai multe generații de facilități de manipulare a fișierelor. Facilitățile originale de intrare/ieșire, care includeau gestionarea de bază a fișierelor, se află în pachetul java.io. Facilități mai cuprinzătoare au fost incluse în noul pachet I/O java.nio. Facilități și mai cuprinzătoare au fost incluse în noul pachet I/O 2 java.nio.file. Ambele pachete au introdus o serie de metode pentru a sprijini controlul mai fin asupra creării fișierelor.

O soluție este crearea unui fișier gol folosind java.io.File.createNewfile(), verificarea stării operației și apoi, în caz de succes, deschiderea fișierului pentru scriere. Această soluție este supusă unei condiții de cursă TOCTOU (time-of-check, time-of-use). Este posibil ca un atacator să modifice sistemul de fișiere după ce este creat fișierul gol, dar înainte de deschiderea fișierului, astfel încât fișierul care este deschis este distinct de fișierul care a fost creat.

Această soluție conformă utilizează metoda java.nio.file.Files.write() cu argumentul CREATE_NEW pentru a crea atomic fișierul și lansează o excepție dacă fișierul există deja. Operația atomică garantează evitarea condiției de cursă.

```java
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        String filename = "example.txt";
        String directoryPath = "/var/www/uploads";
        Path filePath = Paths.get(directoryPath, filename);
        String content = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut ur

        try {
            Files.createDirectories(filePath.getParent());
            Files.write(filePath, content.getBytes(), StandardOpenOption.CREATE_NEW);
            System.out.println("File created successfully with content at " + filePath);
        } catch (IOException e) {
            System.err.println("Error occurred while writing to the file: " + e.getMessa
        }
    }
}
```

```
denissa@ubuntu:~/Desktop/secure-coding-practices/Java/FileCreationOverwriteAfter$  cd /home/den
issa/Desktop/secure-coding-practices/Java/FileCreationOverwriteAfter ; /usr/bin/env /usr/lib/jv
m/jdk-21-oracle-x64/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /home
/denissa/.config/Code/User/workspaceStorage/b2005b83808ca836c42320944af10df1/redhat.java/jdt_ws
/FileCreationOverwriteAfter_52e60bd7/bin App
Error occurred while writing to the file: /var/www/uploads/example.txt
```

ABSENȚA IMPUNERII UNOR DIMENSIUNI LIMITĂ LA DEZARHIVARE + ATAC ZIP-BOMB DoS

O bombă zip este o arhivă rău intenționată, de o dimesniune mică, care perturbă funcționarea normală a dispozitivului odată ce e deschisă. Bombele zip exploatează principiile compresiei datelor, vizând în mod specific modelele de date repetitive pentru a crea o arhivă care se decomprimă la o dimensiune extrem de mare. Dimensiunea afișată pentru astfel de fișiere zip poate fi de numai câțiva kiloocteți.

Bombele zip ar putea încerca să oprească programele antivirus și să lase dispozitivele neprotejate. Un hacker rău intenționat este capabil să ruleze astfel un fișier executabil pe computerul victimei. În timp ce antivirusul analizează arhiva, executabilul ar putea fura cu ușurință date sau pune pe whitelist-ul antivirusului programul malițios, obținând control deplin asupra sistemului.

Bombele zip recursive care includ arhive în alte arhive nu mai sunt de actualitate, iar majoritatea software-urilor moderne, inclusiv programele antivirus, au măsuri pentru detectarea și prevenirea atacurilor cu aceste bombe zip. Aceste măsuri ar putea include limitarea adâncimii decompresiei recursive, monitorizarea modelelor repetitive care indică o bombă zip sau limitarea utilizării resurselor.

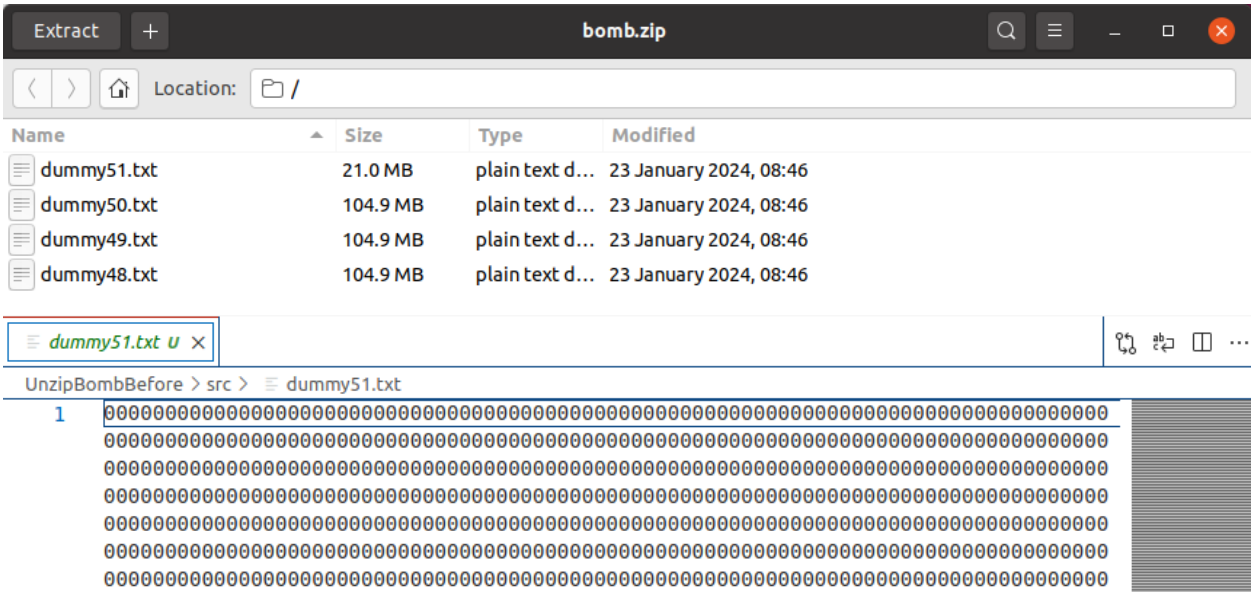
Bombele zip plate decomprimă totul dintr-o dată, fără a trece prin multe runde de decompresie. Acest lucru se realizează printr-un raport de compresie semnificativ mai mare decât ceea ce se observă în mod normal cu fișierele zip. Cel mai bun raport pe care îl poate obține zip este să comprime un fișier de 1032 de ori mai puțin decât dimensiunea sa standard prin algoritmul de compresie DEFLATE. Cu toate acestea, a fost inventată o tehnică pentru ca bombele zip non-recursive să explodeze de peste 28 de milioane de ori (1 kb = > 26,7 GB) într-o singură rundă de decompresie. Bombele non-recursive vor acapara direct resursele sistemului într-o singură rundă de extracție, fără a fi detectate de majoritatea software-urilor antivirus actuale. De aceea, ar trebui acordată o atenție deosebită încărcărilor de fișiere de la utilizatori,

deoarece site-urile neprotejate ar putea suferi atacuri DoS (Denial of Service) din cauza unor astfel de bombe.

Java oferă pachetul java.util.zip pentru compresia datelor cu zip. Oferă clase care permit citirea, crearea și modificarea de fișiere ZIP. O serie de probleme de securitate trebuie luate în considerare la extragerea de fișiere dintr-un fișier ZIP utilizând java.util.zip.ZipInputStream. Numele fișierelor pot conține informații de traversare a căii care pot determina extragerea acestora în afara directorului dorit, frecvent cu scopul de a suprascrie fișierele de sistem existente. O a doua problemă este că procesul de extracție poate provoca un consum excesiv de resurse de sistem, ceea ce poate duce la un atac DoS atunci când utilizarea resurselor este disproporționat de mare în comparație cu datele de intrare. Programele trebuie să limiteze traversarea căii a unor astfel de fișiere și să refuze extragerea datelor dincolo de o anumită limită.

În repository-ul de Github damianrusinek există un script scris în Python care generează o astfel de bombă zip prin crearea de fișiere mari cu date foarte comprimabile, respectiv un șir lung de zerouri, cu potențialul de a se extinde la o dimensiune masivă atunci când sunt decomprimate. Acesta primește dimensiunea necomprimată ca intrare și oferă două moduri: nested și flat.

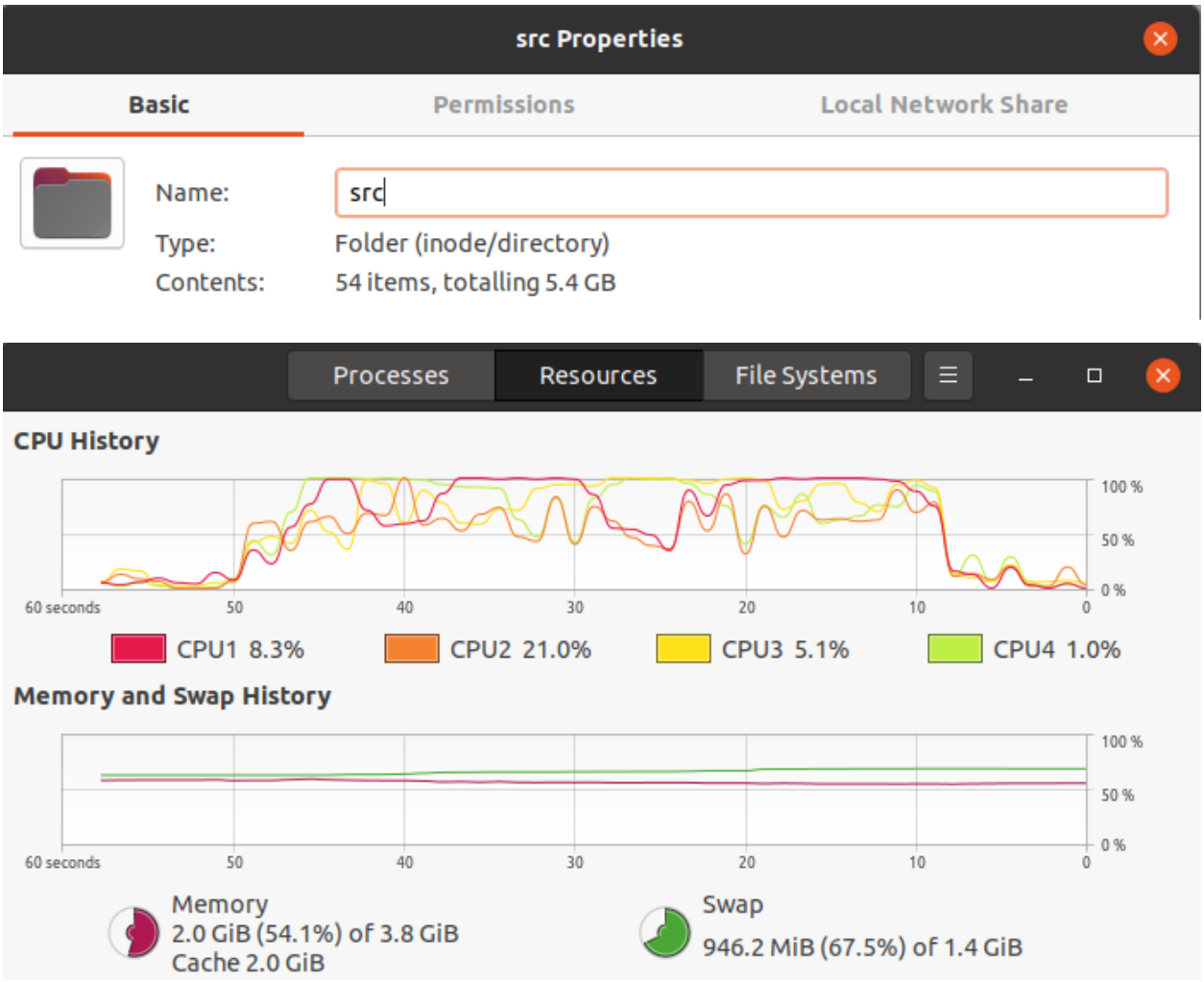
```
denissa@ubuntu:~/Desktop/zip-bomb$ python3 zip-bomb.py flat 5120 bomb.zip
Compressed File Size: 5101.18 KB
Size After Decompression: 5100 MB
Generation Time: 24.11s
```



Acest cod neconform nu reușește să verifice consumul de resurse al fișierului care este dezarhivat. Acesta permite ca operațiunea să se desfășoare până la finalizare sau până la epuizarea resurselor locale.

```java
public class UnsafeUnzipper {

    private static final int BUFFER_SIZE = 512;

    Run | Debug
    public static void main(String args[]) throws IOException {
        Path zipFilePath = Paths.get(System.getProperty(key:"user.dir"), "src" + Fi
        unzip(zipFilePath);
    }

    private static void unzip(Path zipFilePath) throws IOException {
        try (ZipInputStream zis = new ZipInputStream(new BufferedInputStream(
                new FileInputStream(zipFilePath.toFile())))) {
            ZipEntry entry;

            while ((entry = zis.getNextEntry()) != null) {

                Path filePath = zipFilePath.resolveSibling(entry.getName());
                if (!filePath.startsWith(zipFilePath.getParent())) {
                    throw new IOException("Bad zip entry: " + entry.getName());
                }

                System.out.println("Extracting: " + entry);

                if (entry.isDirectory()) {
                    Files.createDirectories(filePath);
                    continue;
                }

                try (BufferedOutputStream dest = new BufferedOutputStream(
                        new FileOutputStream(filePath.toFile()), BUFFER_SIZE)) {
                    byte[] data = new byte[BUFFER_SIZE];
                    int count;
                    while ((count = zis.read(data)) != -1) {
                        dest.write(data, off:0, count);
                    }
                }
                zis.closeEntry();
            }
        }
    }
}
```

```
denissa@ubuntu:~/Desktop/secure-coding-practices/Java/UnzipBombBefore$  /usr/bin/env /usr
/lib/jvm/jdk-21-oracle-x64/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessa
ges -cp /home/denissa/.config/Code/User/workspaceStorage/b2005b83808ca836c42320944af10df1
/redhat.java/jdt_ws/UnzipBombBefore_4914b83e/bin UnsafeUnzipper
Extracting: dummy0.txt
Extracting: dummy1.txt
Extracting: dummy2.txt
Extracting: dummy3.txt
Extracting: dummy4.txt
Extracting: dummy5.txt
```



Codul conform încorporează câteva garanții importante de securitate care pot ajuta la atenuarea riscurilor asociate atât cu bombele ZIP recursive, cât și cu cele non-recursive. Codul stabilește o limită maximă (MAX_UNZIPPED_SIZE, care este de 100 MB) pentru dimensiunea totală a datelor necomprimate. Aceasta este o verificare crucială, deoarece limitează direct impactul unei bombe ZIP. Urmărind totalUnzippedSize și comparându-l cu MAX_UNZIPPED_SIZE, codul garantează că nu decomprimă datele dincolo de un anumit prag, prevenind astfel epuizarea resurselor de stocare sau memorie. Setarea unui număr maxim de fișiere care pot fi extrase (1024) ar putea preveni epuizarea resurselor în cazurile în care o bombă ZIP conține un număr mare de fișiere mici.

```java
public class SafeUnzipper {

    private static final int BUFFER_SIZE = 512;
    private static final long MAX_UNZIPPED_SIZE = 100 * 1024 * 1024; // 100 MB
    private static final int MAX_FILE_COUNT = 1024; // maximum number of files allo

    Run | Debug
    public static void main(String args[]) throws IOException {
        Path zipFilePath = Paths.get(System.getProperty(key:"user.dir"), "src" + Fi
        unzip(zipFilePath);
    }

    private static void unzip(Path zipFilePath) throws IOException {
        try (ZipInputStream zis = new ZipInputStream(new BufferedInputStream(
                new FileInputStream(zipFilePath.toFile())))) {
            ZipEntry entry;
            long totalUnzippedSize = 0;
            int fileCount = 0;

            while ((entry = zis.getNextEntry()) != null) {
                fileCount++;
                if (fileCount > MAX_FILE_COUNT) {
                    throw new IllegalStateException(s:"Too many files to unzip.");
                }

                Path filePath = zipFilePath.resolveSibling(entry.getName());
                if (!filePath.startsWith(zipFilePath.getParent())) {
                    throw new IOException("Bad zip entry: " + entry.getName());
                }

                System.out.println("Extracting: " + entry);

                if (entry.isDirectory()) {
                    Files.createDirectories(filePath);
                    continue;
                }

                totalUnzippedSize += entry.getSize();
                if (totalUnzippedSize > MAX_UNZIPPED_SIZE) {
                    throw new IOException(message:"Unzipped data exceeds allowable
                }

                try (BufferedOutputStream dest = new BufferedOutputStream(
                        new FileOutputStream(filePath.toFile()), BUFFER_SIZE)) {
                    byte[] data = new byte[BUFFER_SIZE];
                    int count;
                    while ((count = zis.read(data)) != -1) {
                        dest.write(data, off:0, count);
                    }
                }
                zis.closeEntry();
            }
        }
    }
}
```

```
denissa@ubuntu:~/Desktop/secure-coding-practices/Java/UnzipBombAfter$  /usr/bin/env /usr/
lib/jvm/jdk-21-oracle-x64/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessag
es -cp /home/denissa/.config/Code/User/workspaceStorage/b2005b83808ca836c42320944af10df1/
redhat.java/jdt_ws/UnzipBombAfter_75eaf93d/bin SafeUnzipper
Extracting: dummy0.txt
Extracting: dummy1.txt
Exception in thread "main" java.io.IOException: Unzipped data exceeds allowable limit.
        at SafeUnzipper.unzip(SafeUnzipper.java:43)
        at SafeUnzipper.main(SafeUnzipper.java:13)
```