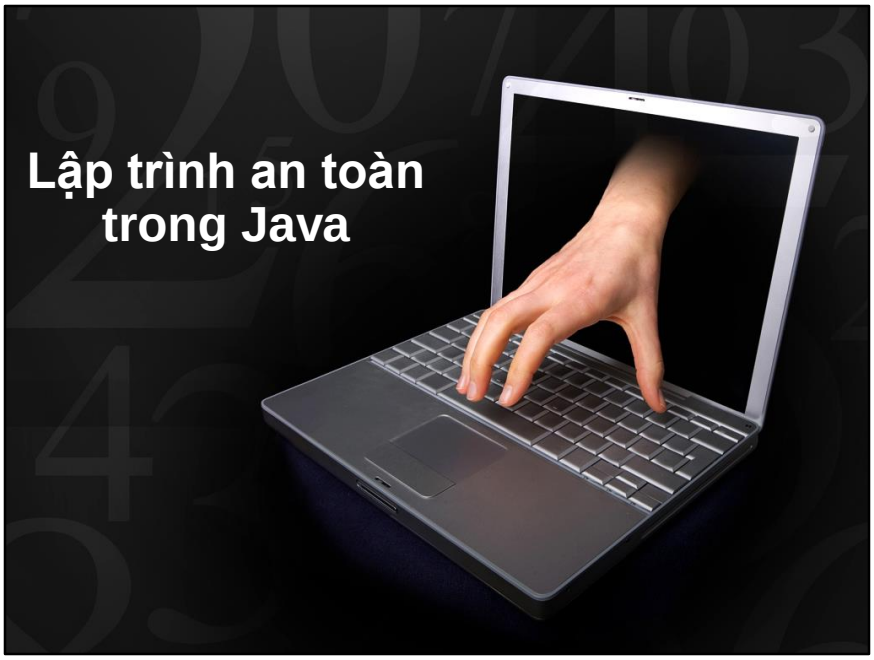


Nội dung

- DoS
- Các dạng DoS

cuu duong than cong . com

Không nên dựa vào
các tính năng bảo mật
để loại bỏ các khiếm khuyết an ninh.

cuu duong than cong . com

2. Từ chối dịch vụ (Denial of Service)

- Đầu vào hệ thống nên được kiểm tra → nguyên nhân tiêu thụ tài nguyên quá mức.
- Tài nguyên thường bị ảnh hưởng:
  - Chu trình CPU (CPU cycle)
  - Bộ nhớ (memory)
  - Không gian đĩa (disk space)
  - Bộ miêu tả tập tin (file descriptor)

Đầu vào hệ thống nên được kiểm tra để chắc rằng nó không là nguyên nhân gây ra việc tiêu thụ tài nguyên quá mức → dẫn đến sự thiếu cân đối trong việc yêu cầu dịch vụ.

Trong một số trường hợp, thực tế không thể đảm bảo rằng đầu vào là hợp lý → vì chúng đòi hỏi phải kết hợp chặt chẽ việc kiểm tra tài nguyên với logic quá trình xử lý dữ liệu.
Các cuộc tấn công gây tiêu tốn tài nguyên quá mức (như DoS liên tục gây lãng phí không gian đĩa) cần được chống lại → hệ thống máy chủ nên mạnh mẽ để chống lại các cuộc tấn công từ bên ngoài.
Đối với máy cá nhân: nếu người dùng đang sử dụng ứng dụng, họ sẽ đóng ứng dụng nếu nó đang sử dụng các nguồn tài nguyên quá mức hoặc bị treo bằng tay.

Bộ miêu tả tập tin: là chỉ số trừu tượng để truy cập 1 tập tin hoặc tài nguyên input/output khác (pipe, network socket).

2.1 – DoS 1: Sử dụng tài nguyên bất thường

- Đồ họa vector có dung lượng lớn.
- Lỗi tràn số nguyên → lỗi kiểm tra kích thước.
- Zip bom: cần hạn chế kích thước dữ liệu khi giải nén.
- Biểu thức Xpath tốn thời gian xử lý bất thường.

…

Chú ý với những hoạt động có thể sử dụng những tài nguyên bất thường:
(Tương tự web: kiểm tra kích thước ảnh, file được upload; kiểm tra loại file.)

- Đồ họa vector: SVG (không giảm chất lượng khi zoom), font.
- Zip bom: 1 tập tin được nén với tỉ lệ rất cao (Zip, Gif, gzip mã hóa nội dung HTTP). Không dựa vào kích thước file nén hay meta-data mà cần hạn chế kích thước dữ liệu khi giải nén.
- Xpath sử dụng biểu thức đường dẫn để lựa chọn node or list node từ tập tin XML, (Xpath Ex định nghĩa pattern).
- Đối tượng được xây dựng bằng cách parse từ text hay binary stream, bộ nhớ tốn gấp nhiều lần so với dữ liệu gốc.
- Mở rộng thực thể XML → tăng đột biến trong quá trình parse.
- Biểu thức chính quy → truy theo vết ngược lại.
- Java Deserialization và Java Beans XML dữ liệu độc → sử dụng bộ nhớ hoặc CPU vô hạn.
- Ghi log quá chi tiết (những hành động không thường) → log file tăng quá mức.
- Loop không xác định có thể bị tạo ra bởi việc parse một vài dữ liệu nhạy cảm mật → chắc rằng mỗi bước lặp tạo ra một vài tiến độ.

2.2 – DoS 2: Giải phóng tài nguyên

- Open file
- Lock (Interface)
- Cấp phát bộ nhớ thủ công

Yêu cầu: hoạt động cấp phát và giải phóng

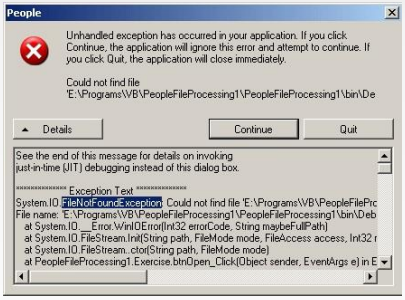
Khi ngoại lệ được đưa ra:

→ Thấy đường dẫn thực thi

→ Giải phóng tài nguyên ngay

Giải phóng tài nguyên trong mọi trường hợp.
Lock: công cụ cho kiểm soát truy cập, chia sẻ tài nguyên trong đa luồng.

2.2 – DoS 2: Giải phóng tài nguyên



2.2 – DoS 2: Giải phóng tài nguyên

Để giảm thiểu lỗi:

- Tối thiểu dư thừa
- Vấn đề xử lý tài nguyên nên được tách riêng

Giải pháp:

- Đặc trưng lambda (Java SE 8)
- Cú pháp try-with-resource
- Sử dụng cấp phát tài nguyên chuẩn
- Chắc rằng output buffer được làm sạch

Đối với những dev có kinh nghiệm cũng thường xuyên xử lý tài nguyên không chính xác.

- Execute Around Method pattern cung cấp 1 cách tuyệt vời để rút trích cặp hoạt động cấp phát và giải phóng. Pattern này được dùng một cách ngắn gọn sử dụng đặc trưng Lamda.
- Try-with-resource: xử lí tự động việc giải phóng nhiều loại tài nguyên.
- Đối với tài nguyên không hỗ trợ đặc trưng nâng cao, sử dụng cấp phát chuẩn và cố gắng sắp xếp quy trình xử lý đủ chặt chẽ.
- Chắc rằng output buffer được làm sạch, nếu flush lỗi -> code nên thoát qua ngoại lệ.

2.2 – DoS 2: Giải phóng tài nguyên

```
long sum = readFileBuffered(new InputStreamHandler() {
        @Override
        public long handle(InputStream in) throws IOException {
                long current = 0;
                for (;;) {
                        int b = in.read();
                        if (b == -1) { return current; }
                        current += b;
                }
        }
});
```


Lamda được hỗ trợ từ Java SE 8.

2.2 – DoS 2: Đặc trưng Lamda

```
long sum = readFileBuffered((InputStream in) -> {
        long current = 0;
        for (;;) {
                int b = in.read();
                if (b == -1) {
                        return current;
                }
                current += b;
        }
});
```


cuu duong than cong . com

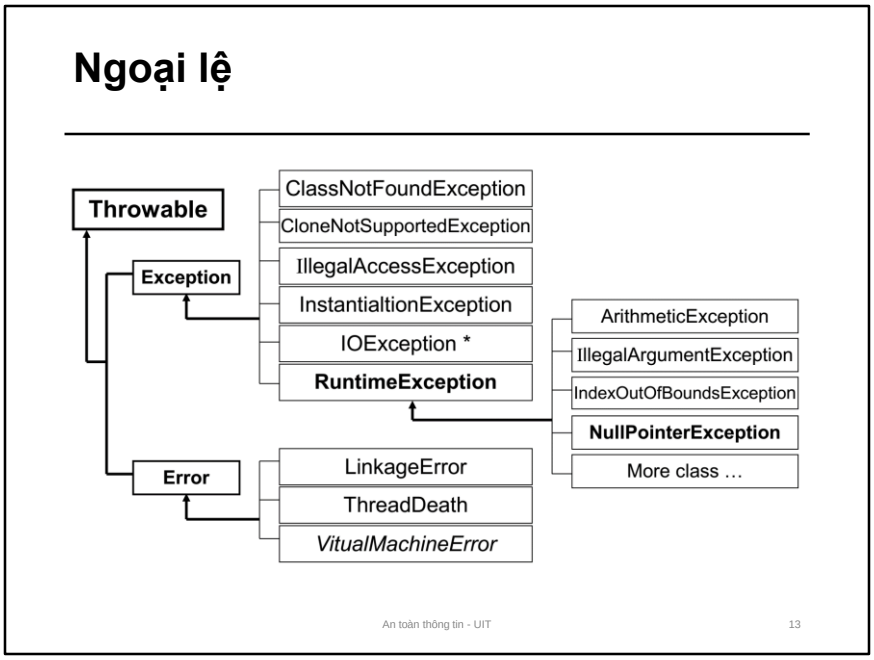
Ngoại lệ

cuu duong than cong . com

Ngoại lệ

- Những lỗi chỉ khi chạy chương trình mới xuất hiện
- Chương trình lập tức dừng lại
- Xuất hiện thông báo lỗi

Ví dụ:
chia 2 số, khi mẫu = 0 thì phát sinh lỗi và đó là một ngoại lệ.

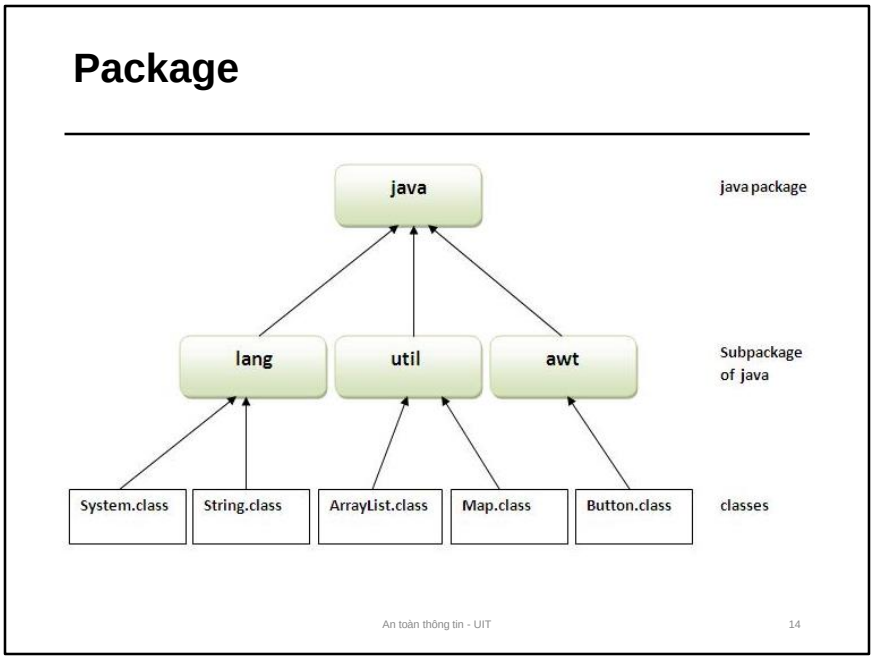


Trong Java Class Throwable xử lý lỗi và ngoại lệ.

Tất cả các class trên hình đều nằm trong gói java.lang, ngoại trừ class IOException là nằm trong gói java.io.

Class Exception: có nhiều ngoại lệ là lớp con của lớp Exception.
• RuntimeErrorException là lớp con của lớp Exception.
• RuntimeErrorException là các ngoại lệ chỉ xảy khi chạy chương trình.
Người lập trình có thể tự tạo các class kế thừa từ class Exception.

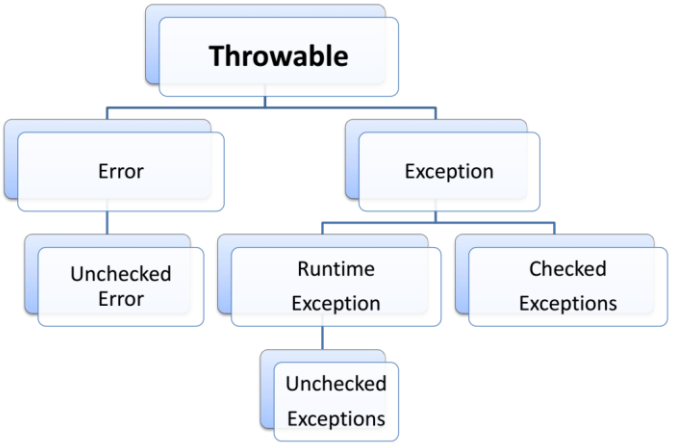
Class Error:
• Chỉ những lỗi nghiêm trọng và không dự đoán trước được như VirtualMachineError, LinkageError, ThreadDead,…
• Các ngoại lệ Error ít được xử lý



Package chứa class, interface, sub-package và được chia thành 2 loại: xây dựng sẵn và người dùng định nghĩa.
Một số package có sẵn: java, lang, util, io, awt, net, sql,…

Java lang chứa các class hệ thống: String, Integer,… và tự import vào hệ thống.

Ngoại lệ



An toàn thông tin - UIT 15

Ngoại lệ '*unchecked*':
• Là các ngoại lệ không cần phải 'catch' khi viết mã.
• Là các class Error, RuntimeException và các lớp con của chúng.

Ngoại lệ '*checked*':
• Là các ngoại lệ phải được 'catch' khi viết mã.
• Là các class còn lại

Ngoại lệ

Một số ngoại lệ ‘checked’:

- ClassNotFoundException
- IOException
- FileNotFoundException
- EOFException

Một số ngoại lệ ‘unchecked’:

- IndexOutOfBoundException
- NullPointerException
- InputMismatchException

cuu duong than cong . com

Ngoại lệ

```
try {
        //Khối lệnh
} catch(…) {
        //Khối lệnh xử lý ngoại lệ
}
```

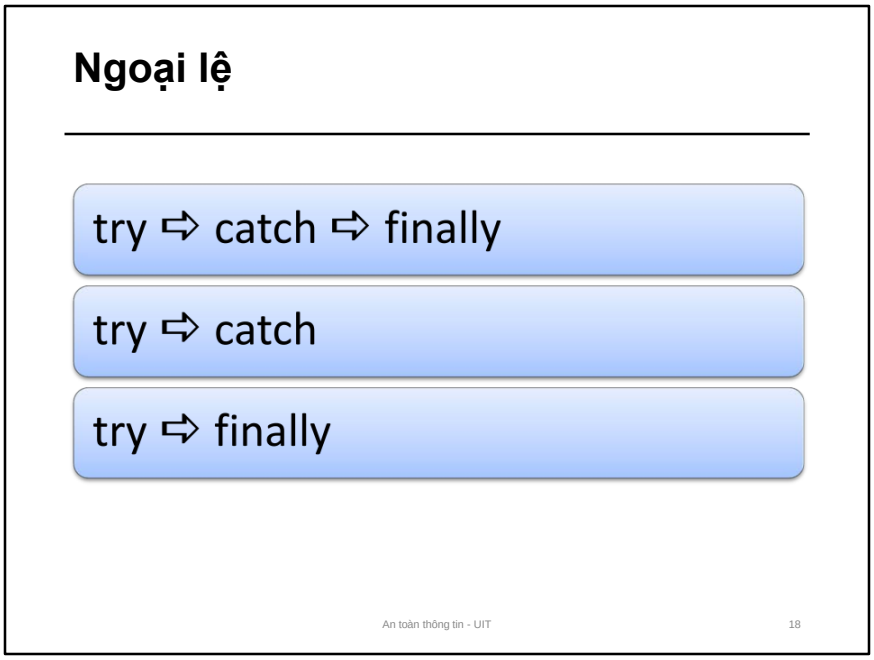
Ví dụ:

```
try {
        c = a/b;
} catch(Exception e) {
        System.out.println("Có lỗi " + e);
}
System.out.println("Sau phép chia!");
```


- Trong một đoạn code có thể có nhiều ngoại lệ xảy ra nên ta sẽ dùng nhiều catch để xử lý các ngoại lệ đó.
- Các lệnh catch thường được viết theo thứ tự xuất hiện của ngoại lệ.

Chú ý:

Tất cả các ngoại lệ sẽ là lớp con của class Exception nên catch Exception sẽ đặt ở vị trí cuối cùng.



Khối finally sẽ chứa một khối mã thực hiện sau khối try/catch.
Khối finally sẽ được thực hiện dù ngoại lệ có xuất hiện hay không. Tuy nhiên, mỗi try sẽ yêu cầu có ít nhất 1 catch hoặc 1 finally.

2.2 – DoS 2: Cú pháp Try-with-resource

```
public  R readFileBuffered(InputStreamHandler handler)
throws IOException {
    try (final InputStream in =
Files.newInputStream(path)) {
        handler.handle(new BufferedInputStream(in));
    }
}
```


Được hỗ trợ từ Java SE 7 – Tự động xử lý giải phóng nhiều loại tài nguyên.
Try-with-resource là 1 loại try có khai báo 1 hoặc nhiều tài nguyên (object) và phải được close sau khi chương trình hoàn thành.

2.2 – DoS 2: Làm sạch bộ nhớ đệm output

```
public void writeFile(OutputStreamHandler handler)
throws IOException {
        try (final OutputStream rawOut =
Files.newOutputStream(path)) {
                final BufferedOutputStream out =
                                new BufferedOutputStream(rawOut);
                handler.handle(out);
                out.flush();
        }
}
```


Nếu flush lỗi thì mã nguồn nên exit qua ngoại lệ.

2.3 – DoS 3: Tránh tràn số nguyên khi kiểm tra giới hạn tài nguyên

Câu hỏi:

- Dung lượng hiện tại: Current GB
- Dung lượng tối đa: Max GB
- Yêu cầu cấp thêm: Extra GB

Thực hiện kiểm tra dung lượng để cấp phát.

Số nguyên thủy và wrap.

Java cung cấp cơ chế kiểm tra giới hạn trên array → hạn chế lượng lớn tấn công tràn số nguyên. Tuy nhiên, 1 số thao tác dùng type số nguyên nguyên thủy vẫn bị tràn số nguyên → cẩn thận khi kiểm tra giới hạn tài nguyên. Đặc biệt quan trọng trên loại tài nguyên ổn định, như không gian đĩa. Khi khởi động lại cũng không giải quyết được vấn đề.

2.3 – DoS 3: Tránh tràn số nguyên khi kiểm tra giới hạn tài nguyên

```
private void checkGrowBy(long extra) {
    if (extra + current < max) {
        // Code xử lý;
    }
}
```

2.3 – DoS 3: Tránh tràn số nguyên khi kiểm tra giới hạn tài nguyên

Current + max → – value

- **Giải pháp 1:** Viết lại điều kiện

```
private void checkGrowBy(long extra) {
        if (extra < 0 || current > max - extra) {
                throw new IllegalArgumentException();
        }
}
```


cuu duong than cong . com

2.3 – DoS 3: Tránh tràn số nguyên khi kiểm tra giới hạn tài nguyên

- **Giải pháp 2:** sử dụng số nguyên lớn

```
private void checkGrowBy(long extra) {
        BigInteger currentBig = BigInteger.valueOf(current);
        BigInteger maxBig     = BigInteger.valueOf(max);
        BigInteger extraBig   = BigInteger.valueOf(extra);

        if (extra < 0 || currentBig.add(extraBig).compareTo(maxBig) > 0) {
                throw new IllegalArgumentException();
        }
}
```


Nếu không quan tâm hiệu suất.

2.3 – DoS 3: Tránh tràn số nguyên khi kiểm tra giới hạn tài nguyên

Chú ý:

Integer.MIN_VALUE == – Integer.MIN_VALUE,
Integer.MIN_VALUE == Math.abs(Integer.MIN_VALUE)

$a \in Z$, $a < 0$ không có nghĩa $-a > 0$

Tương tự cho kiểu: long

Số nguyên bù 2.
Ví dụ: byte [-128, 127]

3. Thông tin mật

- Chỉ được đọc trong ngữ cảnh hạn chế
- Không để lộ → bị giả mạo
- Mã nguồn có đặc quyền không nên được thực thi qua interface có chủ đích

3. 1 – Làm sạch thông tin nhạy cảm từ ngoại lệ

- Ngoại lệ để lộ đường dẫn tập tin
- Lộ tên người dùng hiện tại hoặc thư mục home (Linux)

→ Làm sạch ngoại lệ

- Không gửi ngoại lệ đến người dùng cuối

Lưu ý:

- Có những ngoại lệ sẽ để lộ thông tin dù đã gỡ bỏ thông báo (FileNotFoundException)
- Không nên dựa vào ngoại lệ để bảo mật vì nội dung ngoại lệ có thể thay đổi

Những đối tượng ngoại lệ có thể truyền tải thông tin nhạy cảm.

Ví dụ:

- Nếu một phương thức gọi bộ khởi tạo java.io.FileInputStream để đọc một tập tin cấu hình cơ bản và tập tin đó không tồn tại → Một java.io.FileNotFoundException có chứa đường dẫn tập tin được đưa ra → cho thấy cách bố trí của các tập tin hệ thống.

Có rất nhiều hình thức tấn công đòi hỏi phải biết hoặc đoán vị trí của tập tin.

- Lộ đường dẫn tập tin chứa tên người dùng hiện tại hay thư mục home làm tăng vấn đề bảo mật trong Linux.

Kiểm tra SecurityManager sẽ bảo vệ thông tin này nếu nó được thêm vào trong những thuộc tính hệ thống chuẩn (user.home). Việc lộ thông tin trong ngoại lệ → bypass được kiểm tra.

Những ngoại lệ nội bộ nên được bắt lại và làm sạch trước khi gửi về cho những lời gọi hàm phía trên.

Không gửi ngoại lệ đến người dùng cuối vì nó có thể chứa thông tin nhạy cảm về cấu hình, thông tin nội tại của hệ thống → chỉ gửi khi biết chính xác thứ nó đang chứa.

Đôi khi cũng cần làm sạch ngoại lệ chứa thông tin nhận được từ input của lời gọi hàm. Ví dụ, ngoại lệ liên quan đến việc truy cập file có thể để lộ 1 file có tồn tại hay không → attacker có thể thu thập thông tin bằng cách cung cấp những tên khác nhau như input và phân tích ngoại lệ kết quả.

Không nên dựa vào nogại lệ để bảo mật: ví dụ, version sau của thư viện có thể thêm vào thông tin debug → ứng dụng bộc lộ thông tin thêm dù mã nguồn không đổi.

3. 2 – Không ghi log thông tin có tính nhạy cảm cao

- Không giữ lâu hơn cần thiết
- Không giữ nơi có thể thấy hay qua Search
- Dữ liệu tạm có thể được lưu trong cấu trúc có thể thay đổi, cần làm sạch ngay sau khi sử dụng
- **Lưu ý:** thư viện mức thấp có thể tạo log khi hoạt động

Social Security numbers (SSNs) and passwords cực kì nhạy cảm → không nên giữ lâu hơn cần thiết (không lưu log file) và nơi có thể thấy dù là admin hay có thể tìm ra qua search.

Cấu trúc dữ liệu có thể thay đổi (**mutable data structure)**, như **mảng char**.

Ngoài ra, có thể sử dụng thuật toán mã hóa mạnh, hash password,…

3. 2 – Không ghi log thông tin có tính nhạy cảm cao

String s = “abcde”;

char[] charArray = { 'a', 'b', 'c', 'd', 'e' };

Arrays.fill(charArray, ' ‘);

So sánh:

char[] toCharArray(): chuyển String → char[]

boolean result = Arrays.equals(arr1, arr2);

Class String hỗ trợ phương thức so sánh nhưng không dùng vì string phải đợi cơ chế thu dọn rác.
Sử dụng char[] thay string và làm sạch ngay khi không sử dụng.
Dùng hàm có sẵn để chuyển string → char[] và so sánh 2 array

Char[] có hỗ trợ thao tác copy, copyRange, chuyển về String: String s = new String(charArray).

Lưu ý:
Không convert char[] → string để so sánh.
Dấu ‘ (char) và “ (string) là khác nhau, không giống như php.

3. 3 – Làm sạch bộ nhớ có tính nhạy cảm cao

Thông tin nhạy cảm lưu trong core dump, debugging

→ Nên làm sạch sau khi sử dụng, không đợi cơ chế thu dọn rác

Hạn chế:

- Chất lượng mã nguồn giảm
- Thư viện, máy ảo, hệ điều hành có thể tạo ra nhiều bản copy trong bộ nhớ, thậm chí trên đĩa → không thể xử lý hết

Làm sạch bộ nhớ có tính nhạy cảm cao sau khi sử dụng – hạn chế vòng đời của dữ liệu nhạy cảm.

Tuy nhiên, làm như vậy có thể dẫn đến hậu quả tiêu cực:

- Chất lượng mã nguồn giảm do các rắc rối phụ và cấu trúc dữ liệu có thể thay đổi.
- Không thể làm sạch hết tất cả do thư viện, máy ảo, hệ điều hành có thể tạo ra nhiều bản copy trong bộ nhớ, thậm chí trên đĩa .

Tóm tắt

- **Từ chối dịch vụ**
  - Hoạt động sử dụng tài nguyên bất thường
  - Giải phóng tài nguyên
  - Tránh tràn số khi kiểm tra để cấp phát tài nguyên
- **Thông tin mật**
  - Làm sạch ngoại lệ
  - Không ghi log
  - Làm sạch bộ nhớ có tính nhạy cảm cao

cuu duong than cong . com

Tài liệu tham khảo

Security Manager:

- https://docs.oracle.com/javase/tutorial/essential/environment/security.html
- https://docs.oracle.com/javase/7/docs/api/java/lang/SecurityManager.html

Types, Values, and Variables

- https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html