**Bài thực hành số 1**

**KIỂM THỬ MỘT SỐ LỖ HỔNG PHẦN MỀM**

**Mục tiêu**

Lổ hổng phần mềm là một trong những lỗ hổng an toàn bảo mật phổ biến so với các lỗ hổng khác. Khi kẻ tấn công lợi dụng khai thác lỗ hổng này có thể đem đến thiệt hại ở những mức độ khác nhau tùy thuộc tính nghiêm trọng của lỗ hổng. Phần lớn nguyên nhân lỗ hổng phần mềm xuất hiện là do sự thiếu sót của lập trình viên và đội ngũ kiểm thử. Bài thực hành này được thiết kế để các bạn có thể nhận biết được một số lỗ hổng phần mềm và làm quen với những tình huống kiểm thử hộp trắng đơn giản.

**Môi trường thực hành**

* Máy ảo Server cài đặt hệ điều hành Ubuntu đã triển khai trên phần mềm ảo hóa
* Ngôn ngữ lập trình: C/C++
* Trình biên dịch C/C++: gcc
* Môi trường lập trình Java: Eclipse IDE

**NỘI DUNG THỰC HÀNH**

**1. Một số ví dụ về các lỗ hổng phần mềm**

**1.1. Lỗ hổng tràn bộ đệm (Buffer Overflow)**

Lỗ hổng tràn bộ đệm là một trong 3 lỗ hổng phổ biến và nguy hiểm nhất trên phần mềm. Lỗ hổng này xuất hiện khi lập trình viên không kiểm soát kích thước đầu vào của dữ liệu dần đến vô tình truy cập ra bên ngoài vùng nhớ chứa giá trị đó. Trong phần này, chúng ta sẽ xem xét ví dụ về lỗ hổng tràn bộ đệm trong các chương trình viết bằng ngôn ngữ C/C++ và Java.

* + 1. **Lỗ hổng tràn bộ đệm trong ngôn ngữ C/C++**

*(Các bạn có thể đọc nội dung bộ truyện trang One Piece như một hình thức giải trí vào thời gian rỗi)*

Sau rất nhiều sóng gió, Luffy và các bạn đã đến được hòn đảo Laugh Tale để tiếp cận với kho báu One Piece. Không may cho Luffy là lối vào hòn đảo được bảo vệ bởi một cánh cổng vững chắc. Cánh cổng được cài đặt một chương trình xác thực.

Mở cửa sổ Terminal và thực hiện lệnh **cd /home/admin/bof** để chuyển vào thư mục tương ứng.

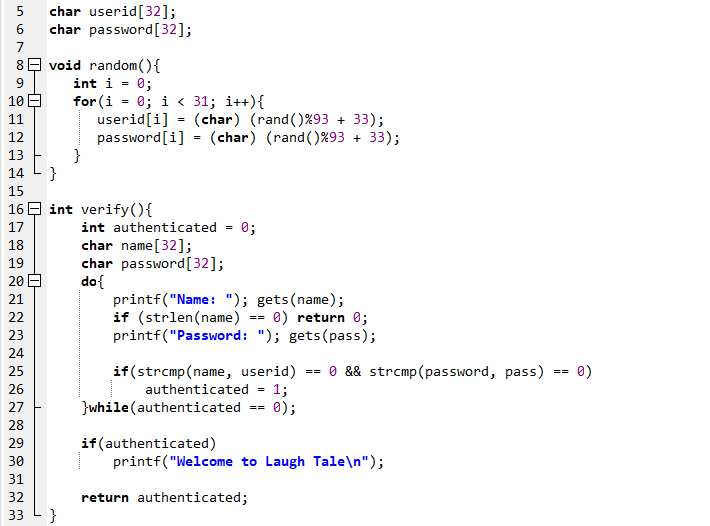
Thực thi chương trình **./vul1**

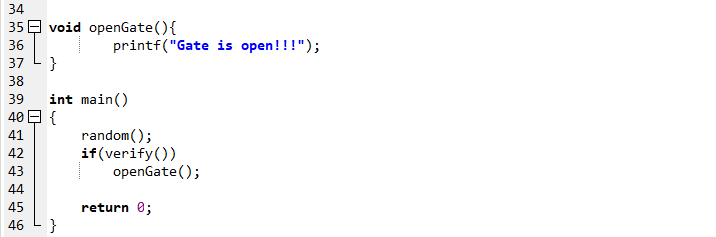
Các bạn có thể thử tùy ý cho tới khi nhập vào dòng trống để thoát ra ngoài khi không đủ kiên nhẫn dò thử thêm.



Với một tay tài năng về công nghệ như Franky trong đội, mã nguồn của chương trình đã được phơi bày.

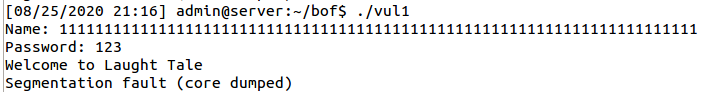
**vul1.c**





Trên dòng 29 của hàm verify() chúng ta thấy, nếu xác thực thành công thì một lời chào mừng được hiện ra. Rẽ nhánh ở dòng 42-43 trong hàm main sẽ thực hiện và cánh cổng được mở. Tuy nhiên, có thể thấy rằng hàm random() sẽ tạo ra các giá trị userid và password ngẫu nhiên. Việc đoán đúng được các giá trị này là một điều vô cùng khó khăn.

Tuy nhiên, có thể thấy rằng từ mã nguồn dòng 20 và 22 của hàm verify(), vì hàm đọc xâu từ bàn phím là gets() không giới hạn số ký tự đọc vào mặc dù khai báo biến là xâu có kích thước 32 ký tự. Vì vậy, chúng ta có thể thử nhập 1 xâu có độ dài lớn hơn nhiều như dưới đây.

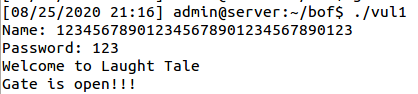


Kết quả cho thấy xâu chào mừng đã được hiển thị, nhưng đồng thời làm chương trình kết thúc ngay. *(Chà, vậy thì chương trình cũng sẽ không mở cánh cổng ra cho Luffy!!!!)*. Có thể coi đây là cách thức đầu tiên để kiểm thử lỗ hổng tràn bộ đệm, ở ví dụ này xảy ra trong hàm verify(). Đối với kẻ tấn công, khai thác lỗ hổng theo hướng này có thể khiến chương trình dừng hoạt động do kích thước dữ liệu đầu vào đã đủ lớn để ghi đè vào giá trị trả về trong stack và gây ra lỗi truy cập bộ nhớ trái phép. Cách khai thác này được coi như là một hình thức tấn công từ chối dịch vụ. Hãy hình dung rằng, thực tế với các dịch vụ mạng có lỗ hổng tương tự như trên, kẻ tấn công chỉ cần gửi một giá trị đầu vào là có thể khiến chương trình máy chủ kết thúc và không thể phụ vụ người dùng.

Như vậy, đây là một vị trí có khả năng là lỗ hổng tràn bộ đệm. Ta sẽ thử kiểm thử bằng một kỹ thuật khác để thấy lỗ hổng có thể bị khai thác mà không làm dừng chương trình. Các biến cục bộ trên bộ nhớ stack có thể được sắp xếp như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| password | name | authenticated | Phần còn lại của stack |
| 32 byte | 32 byte | 4 byte |  |

Có thể thấy biến authenticated nằm sau biến name chiếm 32 byte trong stack. Vì vậy, nếu xâu nhập vào có kích thước từ 33 ký tự trở lên thì ký tự thứ 33 tràn vào vùng nhờ của biến authenticated và làm cho biến này khác 0. Chúng ta thực hiện lại chương trình và thử nhập tên là chuối 123456789012345678901234567890123 còn mật khẩu là bất kỳ.



Kết quả cho thấy giá trị chúng ta nhập vào đã vượt qua được cánh cổng xác thực. Vì biến authenticated đã bị ghi đè giá trị khác 0 nên bất kể kết quả kiểm tra là gì thì vòng lặp 20-27 kết thúc và rẽ nhánh 29-30 đã thực thi.

Trên đây là một cách thức kiểm thử cho thấy nếu khai thác lỗ hổng tràn bộ đệm thì luồng thực thi của chương trình bị thay đổi ngoài mong đợi của lập trình viên.

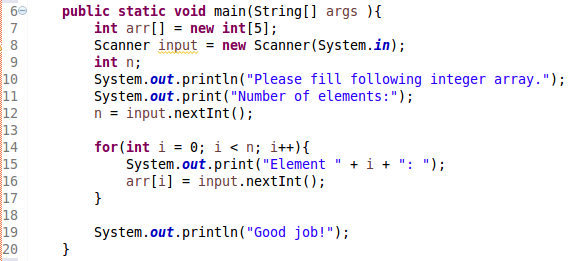
Cách thức khai thác thứ 3 phức tạp nhất và cũng nguy hiểm nhất là kỹ thuật chèn mã thực thi. Kẻ tấn công sẽ cung cấp 1 giá trị đầu vào phù hợp để thực thi chương trình khác theo ý muốn của chúng. Vì lý do phức tạp trong cách thức khai thác, chúng ta sẽ không đề cập đến điều đó ở đây.

* + 1. **Lỗ hổng tràn bộ đệm trong Java**

Một chương trình Java phải chạy trong máy ảo JVM thay vì chạy trực tiếp trên hệ thống. Máy ảo JVM có cơ chế kiểm soát truy cập bộ nhớ riêng thay vì để cho lập trình viên tự do thực hiện điều này giống như ngôn ngữ C/C++. Kẻ tấn công không thể thực hiện các kỹ thuật khai thác để thay đổi luồng thực thi của chương trình. Khi lỗ hổng này trong các chương trình viết bằng Java bị khai thác thì hậu quả ít nghiêm trọng hơn.

Lỗ hổng tràn bộ đệm trong Java thường là dạng truy cập ra ngoài vùng nhớ của biến và gây ra ngoại lệ. Mặc dù không để lại hậu quả nặng nề như đối với lỗ hổng của C/C++, nhưng khi lỗ hổng này bị khai thác dạng tấn công từ chối dịch vụ thì vẫn có những thiệt hại đáng kể xảy ra. Chúng ta cùng xem xét ví dụ sau:

**Vul2.java**

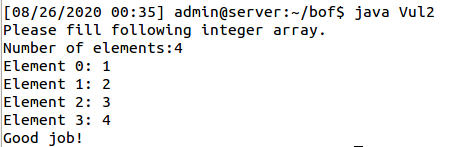


Có thể thấy ở dòng 7, mảng **arr[]** được khai báo với kích thước là 5 phần tử.

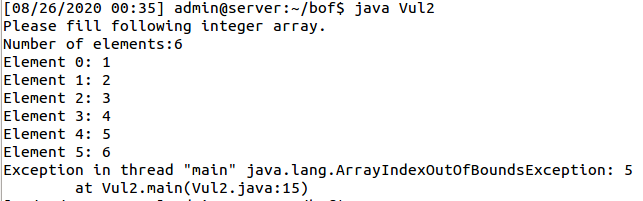
Dòng 12, chương trình nhận từ bàn phím giá trị **n** là số nguyên là số phần tử sẽ nhập vào.

Trên cửa sổ Terminal, trong thư mục làm việc /home/admin/bof, thực hiện lệnh **java Vul2**

* Nếu ta bắt đâu với số phần tử là một giá trị nhỏ, chương trình hoạt động bình thường:



* Tuy nhiên, khi tăng giá trị nhập vào lớn hơn kích thước đã khai báo cho mảng arr[], ta thấy lỗi xuất hiện và chương trình dừng hoạt động bất thường.



Ta xem mã nguồn của chương trình trên:

Vòng lặp 14-17 cho phép người dùng nhập giá trị cho mảng với số phần tử là n. Có thể thấy lỗ hổng xuất hiện là do chương trình không kiểm soát số lượng phần tử mà người dùng sẽ nhập lớn hơn kích thước của mảng. Do đó, trong quá trình thực thi, chương trình sẽ truy cập ra ngoài vùng nhớ của mảng, gây ra ngoại lệ làm dừng chương trình.

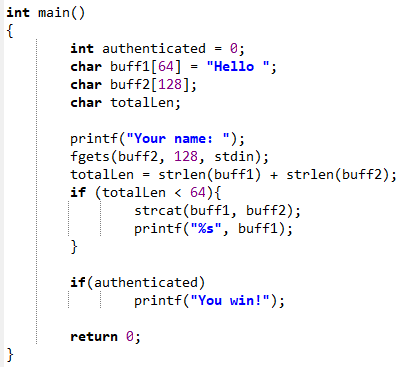
**1.2. Lỗ hổng tràn số nguyên**

Lỗ hổng tràn số nguyên (Interger Overflow) là lỗ hổng mà chương trình không kiểm soát các giá trị gán cho biến số nguyên. Kẻ tấn công có thể lợi dụng lỗ hổng này để gán giá trị này vượt ra khỏi dải biểu diễn của kiểu số nguyên. Tình huống này xảy ra có thể dẫn đến lỗi làm cho chương trình bị gián đoạn hoặc gián tiếp gây ra lỗi tràn bộ đệm.

**1.2.1. Lỗ hổng tràn số nguyên trong C/C++**

Chúng ta cùng xem chương trình sau

**vul3.c**



Chương trình có 2 biến đóng vai trò bộ đệm là buff[1] và buff[2].

Các biến cục bộ trên bộ nhớ stack có thể được sắp xếp như sau:

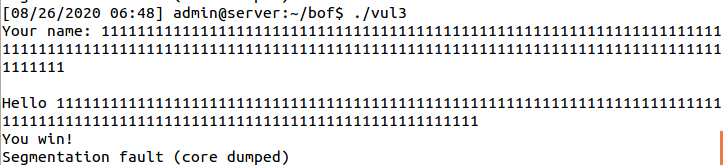
|  |  |  |  |
| --- | --- | --- | --- |
| buff2 | buff1 | authenticated | Phần còn lại của stack |
| 128 byte | 64 byte | 4 byte |  |

Dòng 11 cho phép đọc tối đa 128 ký tự từ bàn phím vào buff2. Do đó không có khả năng xảy ra lỗi tràn bộ đệm trên biến này.

Dòng 12, biến totalLen nhận giá trị là tổng kích thước 2 xâu buff1 và buff2

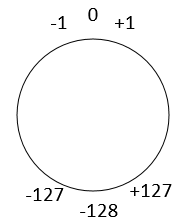
Dòng 14 thực hiện nối xâu buff2 vào xâu buff1 với điều kiện rẽ nhánh là biến totalLen nhỏ hơn 64 là kích thước của bộ đệm buff1. Chúng ta có thể nghĩ rằng với điều kiện này thì dữ liệu không tràn ra khỏi bộ đệm buff1. Có vẻ chương trình sẽ ổn vì biến authenticated không bị thay đổi giá trị và xâu “You win” không được hiển thị.

Chúng ta chạy thử chương trình với lệnh **./vul3**



Kết quả cho thấy lỗi tràn bộ đệm đã xảy ra*(Nếu dùng các công cụ gỡ lỗi như gdb và có kiến thức đầy đủ về bộ nhớ hệ thống máy tính, các bạn có thể nhập vào một xâu với giá trị phù hợp để không xảy ra lỗi truy cập bộ nhớ)*. Điều gì đã xảy ra với chương trình khi chúng ta tưởng như đã kiểm soát được? Hãy xem lại mã nguồn để thấy vấn đề nằm ở đâu. Trước hết, nhìn lại kết quả chúng ta có thể thấy rõ ràng là dữ liệu đã bị tràn ra khỏi biến buff1 và ghi đè lên biến authenticated, tức là điều kiện totalLen < 64 đã được thỏa mãn.

Biến totalLen được khai báo với kiểu char. Trong ngôn ngữ C/C++, kiểu char còn có ý nghĩa là kiểu số nguyên có dấu 8 bit với dải biểu diễn từ -128 tới +127. Chúng ta hãy nhớ lại nguyên tắc biểu diễn số nguyên trong máy tính sẽ tạo thành trục số tròn.

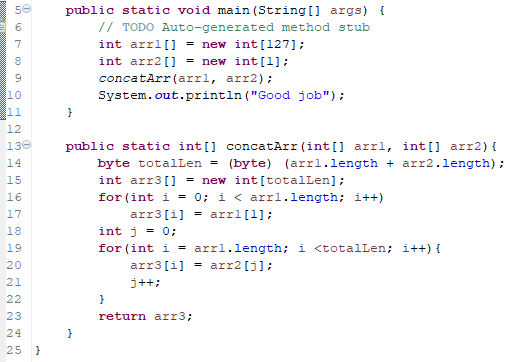


Nếu gán cho totalLen một giá trị lớn hơn +127 thì totalLen sẽ bị tràn sang phần giá trị âm. Như vậy, rõ ràng rằng chỉ cần nhập vào một xâu có độ dài hợp lý, chúng ta sẽ khiến biến totalLen thỏa mãn điều kiện để thực hiện nối xâu.

**1.2.2 Lỗ hổng tràn số nguyên trong Java**

Chúng ta hãy xem một tình huống tương tự xảy ra với ngôn ngữ Java.

**Vul5.java**



Ở dòng 14, biến totalLen nhận một giá trị kiểu **int** bị ép kiểu sang kiểu **byte**. Trong Java, kiểu byte là kiểu số nguyên có dải giá trị từ -128 đến +127. Việc ép kiểu **int** sang **byte** có thể khiến biến kiểu totalLen mang giá trị âm nếu giá trị gán vượt ra khỏi dải biểu diễn. Trong ví dụ này, hàm concatArr được gọi với 2 tham số truyền vào là arr1 và arr2 có tổng kích thước là +128. Biến totalLen = -127 và việc khởi tạo mảng arr3 ở dòng 15 sẽ gây ra ngoại lệ NegativeArraySizeException khiến chương trình bị gián đoạn ngoài ý muốn.

Các bạn có thể lựa chọn các mảng arr1 và arr2 sao cho tổng kích thước của chúng nằm trong đoạn 256 đến 383 để thấy một lỗi khác có thể xảy ra.

**1.3. Lỗ hổng xâu định dạng**

Trong ngôn ngữ C, xâu định dạng được sử dụng trong các hàm vào ra để định dạng dữ liệu đọc từ các thiết bị và hoặc định dạng hiển thị dữ liệu trên thiết bị ra. Lỗ hổng xâu định dạng (Format String) là kiểu lỗ hổng mà các ký tự định dạng trong xâu không phù hợp với dữ liệu cần định dạng. Trong phần này, chúng ta sẽ xem xét lỗ hổng xâu định dạng trong hàm printf() là hàm hiển thị dữ liệu trên thiết bị ra chuẩn là màn hình.

Cú pháp của hàm như sau:

printf(format\_tring, [argument lists])

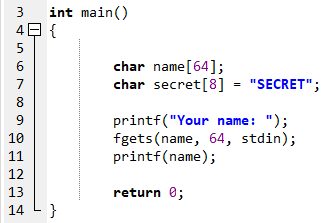
Giả sử khi thực hiện lời gọi printf(“%d%s”, arg1, arg2) thì bộ nhớ stack như sau:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | “%d%s” | arg1 | arg2 |  |
| Vùng nhớ stack của hàm printf | | | | Vùng nhớ stack của hàm gọi hàm print() |

Giá trị của biến arg1 sẽ được hiển thị như số nguyên, còn arg2 được hiển thị như xâu.

Chúng ta xem chương trình ví dụ sau:

**vul4.c**

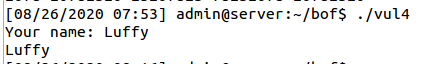


Giả sử rằng xâu secret chứa một dữ liệu bí mật của chương trình (ví dụ như khóa giải mã dữ liệu mật chẳng hạn). Chương trình nhận một xâu từ bàn phím và gán xâu đó cho hàm printf() như là tham số duy nhất. Vì vậy hàm printf sẽ coi xâu này là xâu định dạng. Khi đó, vùng nhớ stack như sau, trong đó giá trị SECRET sẽ nằm đâu đó trong vùng nhớ stack của hàm main.

|  |  |
| --- | --- |
| **name** |  |
| Vùng nhớ stack của hàm printf | Vùng nhớ stack của hàm main() |

Chúng ta hãy xem xét 2 tình huống:

* Tình huống 1: Xâu nhập vào không chứa ký tự định dạng



Trong trường hợp này, hàm printf sẽ hiển thị tất cả các ký tự không phải là ký tự định dạng. Chương trình hoạt động bình thường.

* Tình huống 2: Xâu nhập vào chứa ký tự định dạng



Lúc này ta sẽ có lời gọi hàm printf(“%x”)

|  |  |
| --- | --- |
| **%x** |  |
| Vùng nhớ stack của hàm printf | Vùng nhớ stack của hàm main() |

Trong đó %x là ký tự định dạng để hiển thị 1 số nguyên dưới dạng hexa. Vậy hàm printf() lấy giá trị nào để hiển thị? Nhìn vào hình vẽ trên, có thể thấy 4 byte đầu tiên, trong vùng nhớ stack của hàm main(), tiếp giáp với vùng nhớ của hàm printf() sẽ được lấy ra và hiển thị.

Nếu chúng ta nhập vào một chuối ký tự định dạng “%x %x %x %x…” thì dần dần toàn bộ vùng nhờ stack của hàm main() sẽ bộc lộ, trong đó có những dữ liệu nhạy cảm. Bây giờ chúng ta sẽ thử với đầu vào là chuỗi “%x %x %x %x %x %x %x %x” để đọc ra và hiển thị 8 x 4 = 32 byte.



Mối khối 4 byte cách nhau bởi dấu cách. Chúng ta hãy nhớ rằng mối khối 4 byte này được lưu trữ trong bộ nhớ theo kiểu little-edian với hệ thống máy tính IBM-PC, nghĩa là thứ tự các byte hiển thị ra là thứ tự ngược theo giá trị của chúng. Vì vây, khi sắp xếp lại chúng ta sẽ có chuỗi byte giá trị thực sự là

40 c05afcb7 01 def8ffbf 2f 53454352 4554 25782025

Hãy chuyển dạng biểu diễn hexa này sang mã ASCII. Các bạn có thể viết chương trình đơn giản hoặc dùng công cụ có sẵn như ở trang https://www.convertstring.com/vi/EncodeDecode/HexDecode

Và kết quả chúng ta nhận được như sau:



Dữ liệu bí mật của chương trình đã bị bộc lộ.

**2. Luyện tập thực hành**

Cách thức biên dịch chương trình C từ cửa sổ Terminal:

- Bước 1: Chuyển vào thư mục chứa file mã nguồn với lệnh cd

- Bước 2: Thực hiện lệnh biên dịch theo cú pháp sau:

**gcc *srcfile.c* -o *binfile* -fno-stack-protector**

Trong đó ***srcfile.c*** là file mã nguồn, ***binfile***là file thực thi đã biên dịch được.

**2.1. Bài 1**

Cho chương trình viết bằng ngôn ngữ C giả định chức năng tìm kiếm thông tin của một hệ thống. Nếu người dùng chỉ là khách vãng lai (tài khoản **guest**)thì chỉ hiển thị dữ liệu thông thường. Ngược lại, hiển thị cả dữ liệu nội bộ. Thực hiện các kiểm thử cần thiết để chỉ ra lỗ hổng an toàn bảo mật của chương trình và giải thích các nguy cơ.

**vul6.c**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <stdio.h>  #define GUEST\_ACCOUNT "guest"  #define ADMIN\_ACCOUNT "admin"  **int** main()  {  **char** account[32] = "";  **char** keyword[32];  strcpy(account, GUEST\_ACCOUNT);  printf("What do you want to search? Keyword: "); gets(keyword);  **if** (strcmp(account, GUEST\_ACCOUNT) == 0)  printf("Results: Open Information");  **else** **if** (strcmp(account, ADMIN\_ACCOUNT) == 0)  printf("Results: Secret Information\n");  **return** 0;  } |

**2.2. Bài 2**

Cho chương trình viết bằng ngôn ngữ C để nhập và đọc ra các phần tử của một mảng nguyên dương như sau. Thực hiện các kiểm thử cần thiết để chỉ ra lỗ hổng an toàn bảo mật của chương trình và giải thích các nguy cơ.

**vul7.c**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #include <stdio.h>  **int** main()  {  **int** arr[8];    **int** i = 0, tmp;  printf("----Please fill the array----\n")  **while**(1){  printf("Next elements: "); scanf("%d", &tmp);  **if**(tmp < 0) **break**;  arr[i] = tmp;  i++;  }    printf("----Display array----");  **int** startIdx, endIdx;  printf("Start index: "); scanf("%d", &startIdx);  printf("End index: "); scanf("%d", &endIdx);  **if** (startIdx > 0)  **for**(i = startIdx; i < endIdx; i++)  printf("%d", arr[i]);  **return** 0;  } |

**2.3. Bài 3**

Cho chương trình Java như sau cung cấp chức năng lấy ra một mảng con từ mảng lớn. Thực hiện các kiểm thử cần thiết để xác định những lỗi truy cập bộ nhớ khiến chương trình gián đoạn ngoài ý muốn.

**Vul8.java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | **public** **static** **void** main(**byte**[] args)  {  Scanner scanner = **new** Scanner(System.***in***);    System.***out***.println("Please fill the array");  System.***out***.print("Number of element: ");  **int** n = scanner.nextInt();    **byte**[] input = **new** **byte**[n];  **for**(**int** i = 0; i < n; i++){  System.***out***.print("Element " + i + ": ");  input[i] = scanner.nextByte();  }  System.***out***.println("Extract data from array");  System.***out***.print("Start index: ");  **int** start = scanner.nextInt();  System.***out***.print("End index: ");  **int** end = scanner.nextInt();  **byte**[] subData = *subArray*(input, start, end);    System.***out***.print("Result: ");  **for**(**int** i = 0; i < subData.length; i++)  System.***out***.print("Element " + i + ": " + subData[i]);  }    **public** **static** **byte**[] subArray(**byte**[] source, **int** beginIdx, **int** endIdx)  {  **byte**[] result = **new** **byte**[endIdx - beginIdx];  **int** k = 0;  **for**(**int** i = beginIdx; i < endIdx; i++){  result[k] = source[i];  k++;  }    **return** result;  } |