**Bài thực hành số 2**

**VÁ LỖI MỘT SỐ LỖ HỔNG PHẦN MỀM**

**Mục tiêu**

Sau khi thực hiện kiểm thử lỗ hổng phần mềm, chúng ta cần thực hiện các thao tác vá lỗ hổng để giảm thiểu các mối đe dọa tấn công vào phần mềm. Bài thực hành này được thiết kế để giới thiệu một số kỹ thuật vá lỗ hổng và yêu cầu học viên luyện tập các kỹ thuật này.

**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. Vá lỗ hổng tràn bộ đệm**

Lỗ hổng tràn bộ đệm đến từ sai sót không kiểm soát giá trị dữ liệu đầu vào đẫn đến dữ liệu tràn và ghi đè hoặc đọc được dữ liệu trên vùng nhớ khác. Vì vậy, các kỹ thuật vá lỗ hổng dạng này có 4 dạng:

* Thẩm tra dữ liệu đầu vào(Input Validation): Từ chối xử lý nếu giá trị dữ liệu đầu vào không hợp lệ.
* Chuẩn hóa dữ liệu đầu vào(Input satinization): Xử lý dữ liệu đầu vào để biến thành dữ liệu hợp lệ.
* Canh giữ dữ liệu(Stack Guard): Phát hiện dữ liệu của chương trình bị ghi đè

Hai kỹ thuật đầu được sử dụng dựa trên việc kiểm tra điều kiện với dữ liệu đầu vào. Khi không biết rõ điều kiện này, thường kỹ thuật canh giữ sẽ được sử dụng. Tuy nhiên, kỹ thuật canh giữ chỉ hiệu quả khi áp dụng cho dữ liệu nằm trong vùng nhớ stack. Trên thực tế, người ta có thể kết hợp nhiều kỹ thuật khác nhau.

**1.1. Thẩm tra dữ liệu đầu vào**

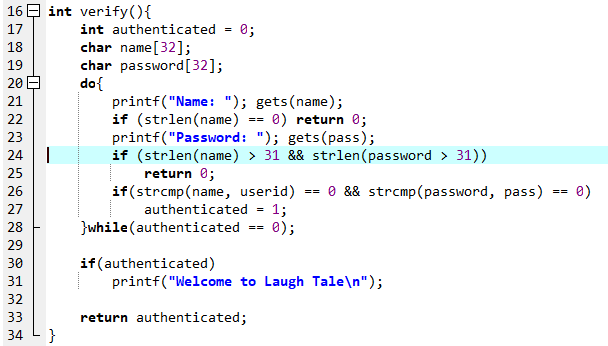
Nguyên lý cơ bản của kỹ thuật này là tạo ra các bộ điều kiện để xác định kích thước dữ liệu và giá trị dữ liệu đầu vào có hợp lệ không. Dữ liệu đầu vào thường sẽ bị từ chối nếu:

* Kích thước lớn hơn kích thước của vùng nhớ (bộ đệm) lưu trữ.
* Giá trị đầu vào gây ra truy cập ngoài vùng nhớ của bộ đệm được phép truy cập ví dụ như chỉ số của mảng, kích thước dữ liệu cần cấp phát

Sau đây chúng ta sẽ viết lại mã nguồn của một số ví dụ trong bài thực hành trước:

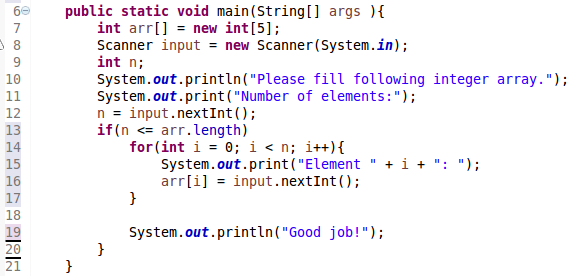
**Chương trình vul1.c**

Điều kiện được thêm vào ở dòng 24 để nếu kích thước xâu người dùng nhập vào lớn hơn kích thước của biến xâu tương ứng thì hàm sẽ trả về giá trị 0(nghĩa là tài khoản không hợp lệ) và kết thúc ngay.



**Chương trình Vul2.java**

Điều kiện được thêm ở dòng 13 để vòng lặp nhập dữ liệu cũ chỉ thực hiện nếu kích thước người dùng nhập vào nhỏ hơn hoặc bằng kích thước mảng dữ liệu.



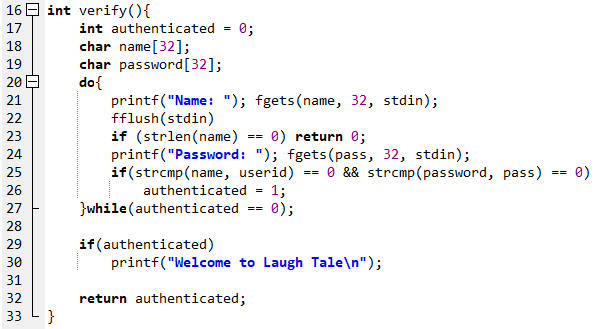
**1.2. Chuẩn hóa dữ liệu đầu vào**

Cách thức chung của kỹ thuật này là làm cho dữ liệu đầu vào trở thành hợp lệ với ngữ cảnh của chương trình. Thông thường có thể thực hiện điều này bằng 2 cách:

* Sử dụng các hàm thay thế cung cấp sẵn cơ chế chuẩn hóa dữ liệu vào
* Thực hiện các thao tác tiền xử lý để chuẩn hóa theo ngữ cảnh của chương trình.

Sau đây chúng ta sẽ viết lại mã nguồn của một số ví dụ trong bài thực hành trước:

**Chương trình** **vul1.c**



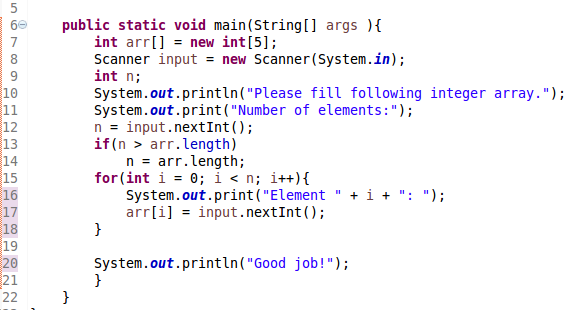
Hàm fgets() đã được sử dụng thay thế cho hàm gets ở dòng 21 và 24. Hàm này có cú pháp như sau:

char \* fgets ( char \* str, int max, FILE \* stream );

Hàm này đọc tối đa **max-1** ký tự từ luồng stream và gán vào xâu str. Khi đọc từ bàn phím thì tham số stream được án là **stdin**.

Với cách gọi hàm như trên, xâu name và pass chỉ nhận tối đa 31 ký tự từ bàn phím. Hàm fflush(stdin) được gọi để xóa bộ đệm bàn phím.

**Chương trình Vul2.java**



Dòng 13 kiểm tra nếu số phần từ người dùng nhập vào lớn hơn kích thước của mảng thì gán lại bằng kích thước mảng.

**1.3. Sử dụng biến canh giữ**

Biến canh giữ được sử dụng để kiểm tra xem dữ liệu có bị tràn ra khỏi bộ đệm hoặc tràn sang vùng nhớ cần bảo vệ hay không. Cách thức này được sử dụng khi khó có thể thực hiện các cách thức trên do chưa xác định được điều kiện hợp lệ của dữ liệu đầu vào. Thực hiện như sau:

- Bước 1: Khai báo các biến canh giữ (canary) bao quanh dữ liệu cần bảo vệ hoặc bộ đệm.

- Bước 2: Gán giá trị ngẫu nhiên cho các biến này trước khi bắt đầu các thao tác gán giá trị đầu vào cho bộ đệm

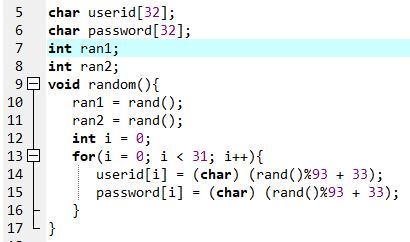
- Bước 3: Kiểm tra lại giá trị các biến canh giữ còn bằng giá trị ngẫu nhiên của bước 2 sau khi truy cập bộ nhớ hay không.

Khi sử dụng biến canh giữ, bộ nhớ stack sẽ như sau:

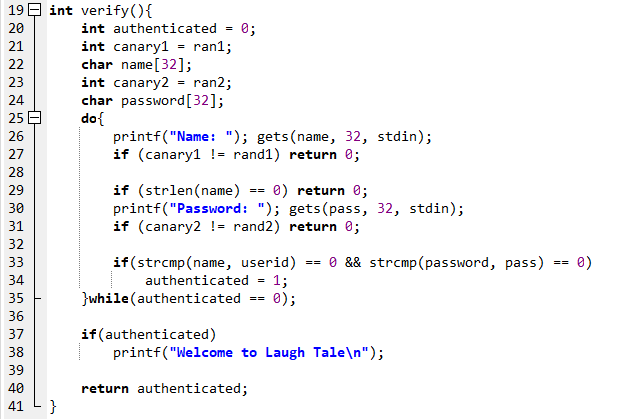
|  |  |  |
| --- | --- | --- |
| Bộ đệm | Biến canh giữ | Vùng nhớ còn lại |

Có thể thấy, nếu bị tấn công, dữ liệu sẽ tràn và ghi đè vào biến canh giữ trước. Số lượng biến canh giữ được sử dụng tùy thuộc vào số lượng bộ đệm và mức độ an toàn cần đạt. Cần chú ý rằng, kẻ tấn công vẫn có xác suất, dù rất nhỏ, nào đó để đoán đúng giá trị ngẫu nhiên ở bước 2. Đây cũng chính là điểm yếu của kỹ thuật này.

Sau đây chúng ta sẽ viết lại mã nguồn chương trình vul1.c.



Ở dòng 7-8 khai báo thêm 2 biến toàn cục ran1 và ran2 là 2 biến chứa giá trị ngẫu nhiên. Hàm random() sẽ khởi tạo giá trị cho 2 biến này.



Trong hàm verify(), chúng ta khai báo 2 biến canh giữ canary1 và canary2 ngay trước các biến đóng vai trò bộ đệm nhận dữ liệu vào là name và password. Khi đó, bộ nhớ stack sẽ như sau:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| password | canary2 | name | canary1 | authenticated | Phần dữ liệu còn lại của stack |

Các biến canh giữ cũng được khởi tạo với giá trị ran1 và ran2.

Dòng 26 kiểm tra điều kiện nếu biến canary1 đã thay đổi so với giá trị khởi tạo, nghĩa là giá trị của nó đã bị ghi đè. Hàm trả về 0(nghĩa là tài khoản không hợp lệ) và kết thúc ngay.

Dòng 29 thực hiện kiểm tra tương tự.

Để vượt qua điều kiện ở dòng 26 và dòng 29 và tấn công thành công, kẻ tấn công phải đoán đúng giá trị ngẫu nhiên ran1 và ran2. Xác suất đoán đúng mỗi giá trị này là 2-32 hay nói cách khác, kẻ tấn công phải chạy thử chương trình 232 lần. Với nhiều trường hợp, việc chạy thử số lần này là không khó khăn. Do đó, để an toàn hơn, chúng ta có thể sử dụng biến canh giữ là một mảng để tăng kích thước giá trị canh giữ, tức là làm cho xác suất đoán đúng của kẻ tấn công giảm xuống. Phần này là bài tập dành cho các bạn suy nghĩ và thực hiện.

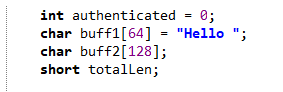
**2. Vá lỗ hổng tràn số nguyên**

Mặc dù là một lỗi tưởng chừng đơn giản nhưng việc vá lỗ hổng tràn số nguyên lại khó hơn và không có cách thức cụ thể. Lý do là để xác định các giá trị có tràn hay không lại phụ thuộc vào ngữ cảnh của chương trình cụ thể. Nguyên tắc chung là thực hiện kiểm thử cẩn thận với các giá trị biên mà có khả năng gây ra tràn số. Nếu việc tràn số xảy ra, cần phải thay thế kiểu dữ liệu có dải rộng hơn. Mặt khác, việc phòng chống lỗi tràn số nguyên có thể bắt đầu từ công đoạn viết chương trình, mà người lập trình viên cần phải cẩn thận với các tình huống ép kiểu.

Sau đây chúng ta sẽ sửa mã nguồn của các chương trình có lỗ hổng tràn số nguyên trong bài thực hành số 1 để làm ví dụ.

**Chương trình vul3.c**

Trong chương trình này, biến totalLen bị tràn do chỉ khai báo là kiểu char(có dải giá trị từ -128 đến 127). Chúng ta có thể chuyển sang dùng kiểu dữ liệu int hoặc short.



Các kiểu dữ liệu này có dải biểu diễn rộng hơn nên không xảy ra tràn số nguyên trong dòng lệnh với mọi giá trị xâu buff1 và buff2. Vì vậy, hacker không thể khai thác lỗ hổng này.

totalLen = strlen(buff1) + strlen(buff2)

**Chương trình Vul5.java**

Trong phương thức concatArray(), chúng ta khai báo lại biến totalLen có kiểu int và bỏ đi toán tử ép kiểu.



**3. Vá lỗi lỗ hổng xâu định dạng**

Việc vá lỗi lỗ hổng xâu định dạng khá đơn giản, chỉ cần tuân thủ nguyên tắc luôn sử dụng ký tự định dạng phù hợp cho mọi giá trị cần hiển thị. Ví dụ, chương trình **vul4.c**, chúng ta chỉ cần sửa lại dòng lệnh hiển thị xâu đầu vào thành **printf(“%s”, name)**.

**2. Luyện tập thực hành**

Thực hiện các kỹ thuật đã được giới thiệu để vá các lỗ hổng cho những chương trình ở bài thực hành số 1. Có thể kết hợp các kỹ thuật khác nhau nếu thấy cần thiết. Sau đó, thực hiện lại các kiểm thử để chắc chắn các lỗ hổng này đã được vá.

**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;  } |

**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;  } |

**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;  } |