**I/ Các kiến thức nền tảng:**

**1. Bộ nhớ stack:**

**a) Thanh ghi (32bit)**

- Khái niệm: Thanh ghi là một phần quan trọng của bộ vi xử lý (CPU), nó được sử dụng để lưu trữ và xử lý các giá trị trung gian trong quá trình thực hiện các lệnh của chương trình

- Mục đích chung: dùng để thực hiện các phép tính toán thông thường (EAX, EBX, ECX). Một thanh ghi mục đích chung khác cần chú ý là ESP hay con trỏ stack

A screenshot of a computer

Description automatically generated

**b) Stack**

- Khái niệm: Ngăn xếp (stack) là một cấu trúc dữ liệu giống như mảng trong bộ nhớ, trong đó dữ liệu có thể được lưu trữ và xóa khỏi vị trí được gọi là đỉnh ngăn xếp. Dữ liệu cần lưu trữ sẽ được push vào ngăn xếp và dữ liệu cần lấy ra sẽ được pop khỏi ngăn xếp. Hoạt động theo nguyên lý cấu trúc LIFO (last in first out)

- Giới hạn của stack được xác định bằng thanh ghi ESP luôn trỏ đến đỉnh của stack

- Các lệnh tác động đến stack như PUSH và POP sử dụng ESP để viết vị trí của stack trong bộ nhớ, Dữ liệu được đưa vào stack với lệnh PUSH; dữ liệu được đưa ra khỏi stack với lệnh POP

**\*\* Câu lệnh:**

– PUSH operand (đẩy dữ liệu vào ngăn xếp)

A diagram of a stack

Description automatically generated

– POP address/register (lấy dữ liệu ra khỏi ngăn xếp)

A diagram of a stack

Description automatically generated

- Theo quy ước, ngăn xếp tăng dần về phía dưới địa chỉ bộ nhớ. Thêm một cái gì đó vào ngăn xếp có nghĩa là đỉnh của ngăn xếp hiện ở mức địa chỉ bộ nhớ thấp hơn

**2. Vị trí của Buffer trên Stack**

- Khi một chương trình trong C/C++ khai báo một buffer (chẳng hạn một mảng), buffer này sẽ được cấp phát trên stack. Ví dụ:

A screenshot of a computer

Description automatically generated

- Khi dữ liệu vượt quá kích thước của buffer, nó sẽ ghi đè lên các vùng nhớ khác trên stack. Điều này có thể bao gồm các giá trị quan trọng như con trỏ khung stack, biến cục bộ khác, hoặc địa chỉ trả về

**3. Địa chỉ trả về**

- Khi một hàm hoàn thành, chương trình sẽ dùng giá trị địa chỉ trả về để biết nơi tiếp tục thực thi. Nếu địa chỉ trả về này bị ghi đè bởi dữ liệu không mong muốn, chương trình sẽ nhảy đến vị trí đó thay vì quay lại vị trí đúng

* Địa chỉ trả về ban đầu có thể là 0x080484a0.
* Nếu buffer bị tràn và ghi đè lên địa chỉ này, địa chỉ trả về có thể là 0x0804dead, dẫn đến lỗi chương trình hoặc thực thi mã độc.

**4. Quản lí bộ nhớ**

- Khi thực thi chương trình, các thành phần khác nhau của chương trình sẽ được ánh xạ vào bộ nhớ một cách có tổ chức

- Đầu tiên, hệ điều hành tạo 1 vùng địa chỉ để chạy chương trình. Vùng địa chỉ này bao gồm các lệnh thực thi của chương trình cũng như các dữ liệu cần thiết

- Tiếp theo, các thông tin được đưa từ file thực thi của chương trình sang vùng địa chỉ vừa tạo. Có 3 dạng segment khác nhau bao gồm: .text, .bss, and .data

- Stack và heap được khởi tạo

**II/ Stack-based buffer overflow là gì  
1. Khái niệm**

- Stack overflows – tràn bộ đệm xảy ra do không có cơ chế kiểm soát giới hạn trên buffer trong ngôn ngữ C/C++. Nói cách khác, ngôn ngữ C và các dẫn xuất của nó không có các chức năng sẵn có để đảm bảo dữ liệu đang được sao chép đến buffer sẽ không vượt quá kích thước giới hạn của buffer => nếu người thiết kế chương trình không lập trình chương trình để kiểm tra input vượt quá kích thước, có thể sẽ lấp đầy bộ đệm, và thậm chí nếu đủ lớn, có thể ghi vượt qua giới hạn của buffer

- Khai thác lỗi Stack-based buffer overflow có thể là hình thức khai thác nguy hiểm nhất và phổ biến nhất để chiếm quyền thực thi mã của một quy trình từ xa. Những cách khai thác này cực kỳ phổ biến cách đây 20 năm

- Trong các năm trở lại đây đã có rất nhiều phương pháp nhằm giảm thiểu các cuộc tấn công stack-based buffer overflow của các nhà phát triển hệ điều hành, nhà phát triển ứng dụng và nhà sản xuất phần cứng, thậm chí còn có những thay đổi đối với các thư viện tiêu chuẩn được sử dụng.

- Các hacker khai thác Buffer Overflow attack bằng cách ghi đè bộ nhớ của các ứng dụng. Việc này sẽ làm thay đổi execution path của chương trình, trigger một response làm ảnh hưởng xấu đến các file trên hệ thống. Hoặc thậm chí là làm lộ thông tin cá nhân của người dung

2. **Cách phát hiện lỗi buffer overflow.**

Công nghệ biên dịch tối ưu để phát hiện lỗi tràn bộ đệm trong chương trình C hoặc C++ dựa trên việc phân tích kích thước dữ liệu trong mã nguồn. Trình biên dịch phải hiểu thông tin từ khai báo biến và các lệnh gọi hàm để tạo ra mã đúng.

Phương pháp ph**ân tích liên thủ tục (interprocedural analysis)** giúp trình biên dịch xác định các lỗ hổng như cố gắng sao chép dữ liệu quá lớn vào bộ đệm nhỏ. Bằng cách xây dựng đồ thị kết nối giữa các hàm, trình biên dịch có thể phát hiện lỗi bảo mật tiềm ẩn, chẳng hạn khi dữ liệu ngoại lai được đưa vào bộ đệm không đủ kích thước. Công nghệ này giúp phát hiện lỗi tràn bộ đệm một cách chính xác và hiệu quả hơn các phương pháp khác, cung cấp cái nhìn sâu sắc về bảo mật mã nguồn.

**3. Cách khai thác (cơ bản)**

- **Tìm lỗ hổng buffer overflow**: Kẻ tấn công sẽ tìm cách phát hiện hàm nào trong chương trình không kiểm tra kích thước của dữ liệu đầu vào và có nguy cơ gây tràn buffer. Các hàm phổ biến dễ bị lỗi này bao gồm gets(), strcpy(), sprintf() trong C vì chúng không giới hạn độ dài của chuỗi đầu vào

- **Tạo dữ liệu đầu vào vượt quá kích thước buffer**: Sau khi tìm ra lỗ hổng, kẻ tấn công sẽ cung cấp dữ liệu có kích thước vượt quá giới hạn của buffer. Ví dụ, nếu buffer chỉ có 10 byte, kẻ tấn công có thể gửi dữ liệu dài 50 byte

- **Ghi đè lên địa chỉ trả về**: Khi buffer bị tràn, dữ liệu sẽ tiếp tục ghi đè lên các vùng nhớ khác, bao gồm cả địa chỉ trả về của hàm. Bằng cách ghi đè chính xác lên địa chỉ này, kẻ tấn công có thể thay đổi nơi mà chương trình sẽ quay lại sau khi hàm kết thúc. Nếu địa chỉ này trỏ đến một đoạn mã độc trong buffer, kẻ tấn công có thể thực thi mã đó

VD: "AAAAAAAAAAAAAAAA\x90\x90\x90\x90\xde\xad\xbe\xef" trong đó:

+ "AAAAAAAAAA..." là dữ liệu để làm tràn buffer.

+ \xde\xad\xbe\xef là địa chỉ mới ghi đè lên địa chỉ trả về.

- **Thực thi mã độc**: Sau khi ghi đè địa chỉ trả về, chương trình sẽ nhảy đến địa chỉ chứa mã mà kẻ tấn công kiểm soát, từ đó thực thi mã độc. Đây có thể là một shellcode nhỏ để mở một shell hoặc thực hiện các hành động khác

**\*\* Các kĩ thuật nâng cao hơn: Return-to-libc Attack và Return Oriented Programming (ROP)**

**4. Biện pháp khắc phục**

- **Kiểm tra giới hạn kích thước**: Tránh dùng các hàm không an toàn, thay thế bằng các hàm có cơ chế kiểm soát: strncpy() thay vì strcpy(), fgets() thay vì gets(), snprintf() thay vì sprintf()

- **Sử dụng bộ nhớ heap thay vì stack:** Thay vì sử dụng các buffer trên stack với kích thước cố định, lập trình viên có thể sử dụng bộ nhớ heap (dynamic memory allocation) với kích thước linh động hơn để quản lý dữ liệu lớn. Điều này giúp tránh việc tràn buffer trên stack

- **Sử dụng** **Address Space Layout Randomization (ASLR)**: Kỹ thuật này ngẫu nhiên hóa địa chỉ của các vùng bộ nhớ (như stack, heap, thư viện hệ thống) mỗi khi chương trình chạy. Điều này làm cho việc dự đoán địa chỉ các hàm hoặc buffer trở nên khó khăn hơn đối với kẻ tấn công

- **Sử dụng Data Execution Prevention (DEP):** DEP là một kỹ thuật ngăn chặn việc thực thi mã trong các vùng bộ nhớ không đáng tin cậy, như stack hoặc heap. Điều này có nghĩa là ngay cả khi kẻ tấn công ghi mã độc vào buffer, chương trình sẽ không thể thực thi mã đó

DEMO

Code:

#include <signal.h>

#include <stdio.h>

#include <string.h>

int main(){

char realPassword[20];

char givenPassword[20];

strncpy(realPassword, "ddddddddddddddd", 20);

gets(givenPassword);

if (0 == strncmp(givenPassword, realPassword, 20)){

printf("SUCCESS!\n");

}else{

printf("FAILURE!\n");

}

raise(SIGINT);

printf("givenPassword: %s\n", givenPassword);

printf("realPassword: %s\n", realPassword);

return 0;

}

A screen shot of a computer program

Description automatically generated

LINK: https://www.rapid7.com/blog/post/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/

DEMO:

A screenshot of a computer program

Description automatically generated