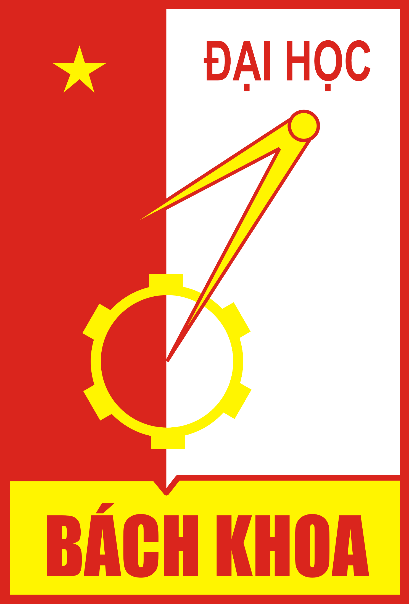
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



**BÁO CÁO AN TOÀN VÀ BẢO MẬT THÔNG TIN**

***Đề tài*: Lỗ hổng phần mềm xây dựng trên ngôn ngữ C**

***GIẢNG VIÊN HƯỚNG DẪN:*** ***Đỗ Văn Uy***

***NHÓM SINH VIÊN THỰC HIỆN***:

***1. Roãn Văn Thụ MSSV: 20144416***

***2. Nguyễn Văn Ngọc MSSV: 20143223***

***3. Nguyễn Văn Tuân MSSV: 20144838***

***4. Trần Nam Anh MSSV: 20140235***

Hà Nội, tháng 4 năm 2017

MỤC LỤC

**Lời nói đầu 3**

1. **Tràn bộ đệm 4**

**1.1 Tràn bộ đệm trên ngăn xếp ( Stack overflow) 5**

**1.2 Tràn bộ đệm trên Heap (Heap overflow) 7**

**1.3 Chống tràn bộ đệm 9**

**2. Định dạngchuỗi (Format String) 11**

**2.1 Một số khái niệm 11**

**2.2 Khai thác lỗ hổng 12**

**2.2.1** **Gây sụp đổ chương trình. 14**

**2.2.2 Xem nội dung Stack. 14**

**2.2.3 Truy cập bộ nhớ bất kỳ. 14**

**2.2.4 Ghi vào vùng nhớ bất kỳ bằng định dạng %n 15**

**2.3 Cách phòng chống lỗi định dạng chuỗi. 16**

**3. Lỗi Integer 16**

**3.1 Một số khái niệm 16**

**3.2 Cách khắc phục và lợi ích việc khắc phục 18**

**4. Con trỏ treo (Dangling poiter) 18**

**4.1 Khái niệm** : **18**

**4.1.1 Dangling poiter 19**

**4.1.2Vấn đề với Dangling Pointer 19**

## **4.2 Nguyên nhân của con trỏ tre ( Dangling Pointer ) trong C 20**

**4.3 Giải quyết vấn đề Dangling Pointer 21**

Lời nói đầu

**Ngôn ngữ lập trình C** là một ngôn ngữ [mệnh lệnh](https://vi.wikipedia.org/w/index.php?title=L%E1%BA%ADp_tr%C3%ACnh_ki%E1%BB%83u_m%E1%BB%87nh_l%E1%BB%87nh&action=edit&redlink=1) được phát triển từ đầu [thập niên 1970](https://vi.wikipedia.org/wiki/Th%E1%BA%ADp_ni%C3%AAn_1970) bởi [Dennis Ritchie](https://vi.wikipedia.org/wiki/Dennis_Ritchie) để dùng trong [hệ điều hành](https://vi.wikipedia.org/wiki/H%E1%BB%87_%C4%91i%E1%BB%81u_h%C3%A0nh) [UNIX](https://vi.wikipedia.org/wiki/Unix). Từ đó, ngôn ngữ này đã lan rộng ra nhiều hệ điều hành khác và trở thành một những ngôn ngữ phổ dụng nhất. C là ngôn ngữ rất có hiệu quả và được ưa chuộng nhất để viết các [phần mềm hệ thống](https://vi.wikipedia.org/wiki/Ph%E1%BA%A7n_m%E1%BB%81m_h%E1%BB%87_th%E1%BB%91ng), mặc dù nó cũng được dùng cho việc viết các [ứng dụng](https://vi.wikipedia.org/wiki/Ph%E1%BA%A7n_m%E1%BB%81m_%E1%BB%A9ng_d%E1%BB%A5ng). Ngoài ra, C cũng thường được dùng làm phương tiện giảng dạy trong [khoa học máy tính](https://vi.wikipedia.org/wiki/Khoa_h%E1%BB%8Dc_m%C3%A1y_t%C3%ADnh) mặc dù ngôn ngữ này không được thiết kế dành cho người nhập môn.

C đã được tạo ra với một mục tiêu là làm cho nó thuận tiện để viết các chương trình lớn với số lỗi ít hơn trong [mẫu hình](https://vi.wikipedia.org/wiki/M%E1%BA%ABu_h%C3%ACnh) [lập trình thủ tục](https://vi.wikipedia.org/wiki/L%E1%BA%ADp_tr%C3%ACnh_th%E1%BB%A7_t%E1%BB%A5c) mà lại không đặt gánh nặng lên vai người viết ra [trình dịch](https://vi.wikipedia.org/wiki/Tr%C3%ACnh_bi%C3%AAn_d%E1%BB%8Bch) C, là những người bề bộn với các đặc tả phức tạp của ngôn ngữ.

Một câu nói phổ biến được người ta lập lại nhiều lần của một nhà thiết kế trình dịch [Bjarne Stroustrup](https://vi.wikipedia.org/wiki/Bjarne_Stroustrup), người sáng lập ra [C++](https://vi.wikipedia.org/wiki/C%2B%2B), là "*C makes it easy to shoot yourself in the foot.*" (tạm dịch: "C làm cho việc bạn tự hại bạn trở nên dễ dàng") . Nói cách khác, C cho phép nhiều phép toán không mong muốn trong một cách tổng quát, và do đó, nhiều lỗi đơn giản đã được tạo ra bởi một người lập trình mà chúng lại không thể phát hiện qua trình dịch hay ngay cả không phát hiện ra trong lúc thi hành. Điều này là nguyên nhân của một số chương trình có các ứng xử không lường trước được và có các lỗ hổng về an toàn.

Một lý do của các vấn đề nêu trên là để tránh cho cái giá quá cao phải trả cho việc kiểm soát (lỗi) ở thời gian dịch và thời gian thi hành. Một lý do khác là sự đòi hỏi để giữ C được càng hiệu quả và càng uyển chuyển càng tốt. Bài báo cáo này sẽ tìm hiểu về các lỗ hổng phần mềm được xây dựng trên ngôn ngữ C, các cách tấn công lỗ hổng và biện pháp khắc phục.

1. **Tràn bộ đệm**

Trong các lĩnh vực [an ninh máy tính](https://vi.wikipedia.org/w/index.php?title=An_ninh_m%C3%A1y_t%C3%ADnh&action=edit&redlink=1) và [lập trình](https://vi.wikipedia.org/wiki/L%E1%BA%ADp_tr%C3%ACnh_m%C3%A1y_t%C3%ADnh), một lỗi tràn bộ nhớ đệm hay gọi tắt là lỗi tràn bộ đệm là một lỗi lập trình có thể gây ra một [ngoại lệ](https://vi.wikipedia.org/w/index.php?title=X%E1%BB%AD_l%C3%BD_ngo%E1%BA%A1i_l%E1%BB%87&action=edit&redlink=1) truy nhập bộ nhớ máy tính và chương trình bị kết thúc, hoặc khi người dùng có ý phá hoại, họ có thể lợi dụng lỗi này để phá vỡ an ninh hệ thống.

Lỗi tràn bộ đệm là một điều kiện bất thường khi một [tiến trình](https://vi.wikipedia.org/w/index.php?title=Ti%E1%BA%BFn_tr%C3%ACnh_(t%C3%ADnh_to%C3%A1n)&action=edit&redlink=1) lưu [dữ liệu](https://vi.wikipedia.org/wiki/D%E1%BB%AF_li%E1%BB%87u) vượt ra ngoài biên của một [bộ nhớ đệm](https://vi.wikipedia.org/w/index.php?title=B%E1%BB%99_nh%E1%BB%9B_%C4%91%E1%BB%87m&action=edit&redlink=1) có chiều dài cố định. Kết quả là dữ liệu đó sẽ đè lên các vị trí [bộ nhớ](https://vi.wikipedia.org/wiki/B%E1%BB%99_nh%E1%BB%9B) liền kề. Dữ liệu bị ghi đè có thể bao gồm các bộ nhớ đệm khác, các biến và dữ liệu điều khiển luồng chạy của chương trình (*program flow control*).

Các lỗi tràn bộ đệm có thể làm cho một tiến trình đổ vỡ hoặc cho ra các kết quả sai. Các lỗi này có thể được kích hoạt bởi các dữ liệu vào được thiết kế đặc biệt để thực thi các đoạn mã phá hoại hoặc để làm cho chương trình hoạt động một cách không như mong đợi. Bằng cách đó, các lỗi tràn bộ đệm gây ra nhiều [lỗ hổng bảo mật](https://vi.wikipedia.org/w/index.php?title=L%E1%BB%97_h%E1%BB%95ng_b%E1%BA%A3o_m%E1%BA%ADt&action=edit&redlink=1) (*vulnerability*) đối với phần mềm và tạo cơ sở cho nhiều thủ thuật [khai thác](https://vi.wikipedia.org/w/index.php?title=Khai_th%C3%A1c_(an_ninh_m%C3%A1y_t%C3%ADnh)&action=edit&redlink=1) (*exploit*). Việc [kiểm tra biên](https://vi.wikipedia.org/w/index.php?title=Ki%E1%BB%83m_tra_bi%C3%AAn&action=edit&redlink=1) (*bounds checking*) đầy đủ bởi lập trình viên hoặc [trình biên dịch](https://vi.wikipedia.org/wiki/Tr%C3%ACnh_bi%C3%AAn_d%E1%BB%8Bch) có thể ngăn chặn các lỗi tràn bộ đệm.

Một lỗi tràn bộ nhớ đệm xảy ra khi [dữ liệu](https://vi.wikipedia.org/wiki/D%E1%BB%AF_li%E1%BB%87u) được viết vào một bộ nhớ đệm, mà do không [kiểm tra biên](https://vi.wikipedia.org/w/index.php?title=Ki%E1%BB%83m_tra_bi%C3%AAn&action=edit&redlink=1) đầy đủ nên đã ghi đè lên vùng bộ nhớ liền kề và làm hỏng các giá trị dữ liệu tại các [địa chỉ bộ nhớ](https://vi.wikipedia.org/w/index.php?title=%C4%90%E1%BB%8Ba_ch%E1%BB%89_b%E1%BB%99_nh%E1%BB%9B&action=edit&redlink=1) kề với vùng bộ nhớ đệm đó. Hiện tượng này hay xảy ra nhất khi sao chép một [xâu ký tự](https://vi.wikipedia.org/w/index.php?title=X%C3%A2u_k%C3%BD_t%E1%BB%B1&action=edit&redlink=1) từ một bộ nhớ đệm này sang một vùng bộ nhớ đệm khác.

Trong ngôn ngữ C, các chuỗi (string) hay buffer được thể hiện như sau: Con trỏ sẽ chỉ vào byte đầu tiên của chuỗi hay buffer đó, và xác định được điểm kết thúc khi con trỏ trỏ trỏ đến một byte null => không xác định được độ dài của đối tượng nhập vào => có thể copy 1 buffer có độ dài lớn vào buffer có độ dài nhỏ hơn gây tràn bộ nhớ (buffer overflow).

Mã nguồn ví dụ:

#include<stdio.h>

int main(){

int a[8];

a[9] = 1;

printf(“Hello World\n”);

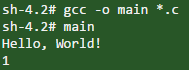
printf(“%d\n”,a[9]);

return 0;

}

Mảng a chỉ có 8 phần tử, nhưng vẫn có thể truy cập và sửa đổi giá trị tương ứng với phần tử thứ 10 trong mảng a (vị trí vùng nhớ thuộc về đối tượng khác).

Kết quả:



### Có hai kiểu tràn bộ đệm chính: Tràn bộ đệm trên stack và tràn bộ đêm trên heap.

### Stack và heap là địa điểm cung cấp các biến khi chạy, một chương trình. Biến được lưu trữ trong stack hoặc heap khi chương trình có nhu cầu sử dụng. Stack là nơi lưu trữ tĩnh của không gian bộ nhớ. Heap là nơi nơi lưu trữ động của không gian bộ nhớ.

### 1.1 Tràn bộ đệm trên ngăn xếp ( Stack overflow)

Bộ đệm (stack) là một vùng nhớ liên tục có kích thước giới hạn ***thường*** được dùng để lưu trữ các giá trị tạm thời của các chương trình con. Trong miền bộ nhớ của một process, đáy stack bắt đầu từ một vị trí nhất định trong bộ nhớ, đỉnh stack thay đổi theo thời gian và được trỏ tới bởi stack pointer (**SP**). Giá trị của biến SP nằm trong một thanh ghi (**ESP**) để truy cập nhanh. Stack chứa vài stack frames.

Khi một chương trình mẹ *ABC* gọi thực hiện một chương trình con *abc*, hệ điều hành sẽ cất địa chỉ quay lui (lệnh tiếp theo của chương trình *ABC*, ngay sau lời gọi *abc*) vào ngăn xếp hệ thống; sau đó các dữ liệu tạm thời của *abc* (tức là các biến nhớ cục bộ của nó) cũng được bố trí tiếp tục trên ngăn xếp này. Bản thân các tham số truyền từ *ABC* cho *abc* cũng sẽ được coi như các biến cục bộ ẩn và lưu giữ trên ngăn xếp này. Khi chương trình con *abc* kết thúc, hệ điều hành sẽ giải phóng phần bộ nhớ tạm lưu các biến cục bộ của nó, rồi lấy ra địa chỉ quay lui, và chuyển điểu khiển về cho đoạn mã lệnh xác định bởi địa chỉ này (tức là lệnh thực thi tiếp theo trong ABC). Như vậy nếu như một trong các biến cục bộ mà bị “tràn”, tức là dữ liệu nhập vào đó lớn hơn dự kiến và tràn sang các địa chỉ bộ nhớ tiếp theo, thì có khả năng chính ô chứa địa chỉ quay lui nói trên có thể bị ghi đè. Từ đó hệ điều hành có thể chuyển điều khiển tới một đoạn mã lệnh khác (hoặc một đoạn dữ liệu bất kỳ), gây ra mất điều khiển hoặc sai hỏng.

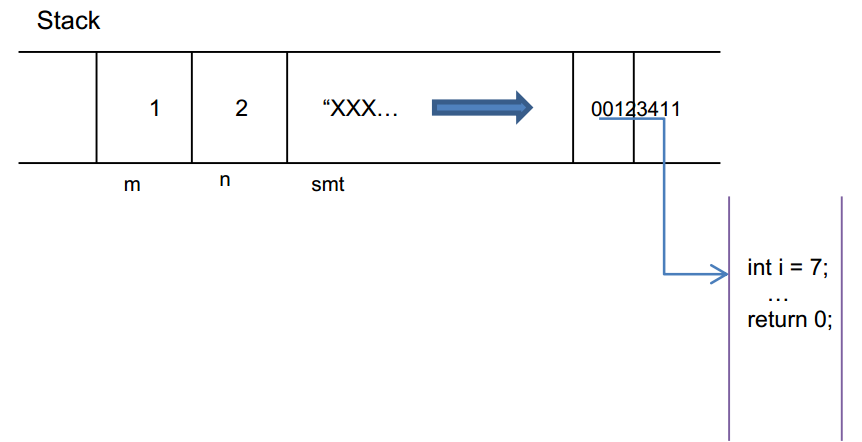


*Xem xét một ví dụ đơn giản sau về lỗi tràn bộ đệm:*  
void func(){  
 int m=1,n=2;  
 smt[126]; //cấp phát bộ nhớ độ dài 126 bytes trên bộ nhớ đệm cho biến a  
 printf(“enter a string\n”);  
 scanf(“%s”,smt); // sao chép giá trị của chuỗi ký tự nhập vào vùng nhớ đệm của biến a  
}

Trong ví dụ trên đây, ta thấy rằng nếu người dùng nhập vào chuỗi ký tự  
“XXX…XXX” (127 chữ X) thì chữ X sau cùng sẽ tràn ra khỏi vùng bộ nhớ đệm của  
biến *smt*.

Tiếp theo ta xét một đoạn chương trình chính sẽ thực hiện lời gọi hàm *func* nói  
trên.  
int main(){  
 …  
 func();  
 int i = 7;  
 …  
 return 0;  
}

Khi hàm *main()* thực hiện lệnh gọi hàm *func();* trước khi mã lệnh *func* được  
thực hiện, hệ điều hành cất địa chỉ lệnh thực hiện tiếp theo (tức lệnh gán giá trị *i=7*)  
vào ngăn xếp. Giả sử địa chỉ bộ nhớ của lệnh gọi hàm func() là 0012340C và lệnh thực  
hiện lệnh i=7 là 00123411. Khi thực hiện lệnh gọi hàm func() thì tình trạng bộ nhớ đệm. sẽ như phản ánh ở hình vẽ dưới, trong đó địa chỉ quay về hàm main(), 00123411, sẽ được lưu vào vùng nhớ đệm ngay phía dưới phần bộ nhớ lưu biến *smt* của hàm func(). Vì vậy, khi thực hiện hàm *scanf* trong hàm *func* nếu người dùng nhập vào *smt* một chuỗi ký tự dài hơn 126 ký tự, thì các ký tự thừa sẽ đè lên địa chỉ quay về ô chứa giá trị 00123411 tạo thành một địa chỉ quay về mới. Kẻ tấn công có thể lựa chọn giá trị nhập vào sao cho địa chỉ quay về là theo ý của kẻ tấn công.

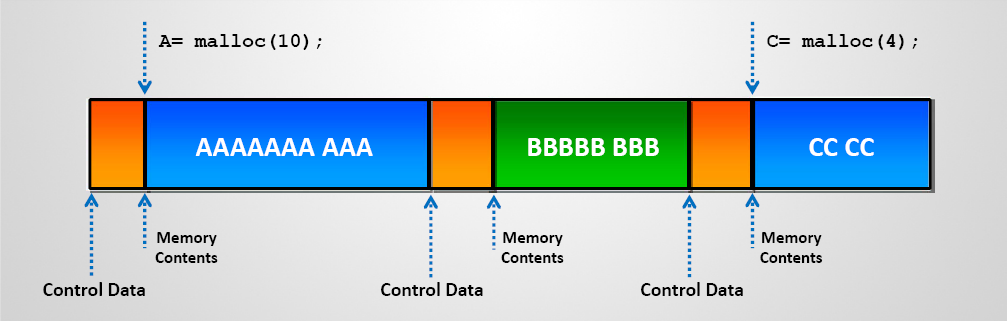


Cơ chế tổng quát này có thể bị lợi dụng bởi kẻ tấn công như sau. Kẻ địch sẽ cố  
tình tìm cách chiếm quyền điểu khiển, tức là làm cho hệ điều hành chuyển điều khiển  
đến một đoạn mã độc có sẵn trong bộ nhớ. Lỗi không kiểm soát bộ đệm (của một lập  
trình viên bất cẩn) có thể bị lợi dụng, kẻ địch sẽ tìm cách nhập dữ liệu sao đó để một  
biến cục bộ tràn vào ô nhớ chứa địa chỉ quay lui. Hơn nữa, giá trị ghi đè này là một địa  
chỉ tính toán trước. Chẳng hạn, hãy tưởng tượng, kẻ địch khôn ngoan có thể tìm cách  
đưa vào bộ nhớ một đoạn mã độc, ngụy trang như một đoạn dữ liệu (bộ nhớ dữ liệu có thể có những giá trị bất kỳ và bình thƣờng không bao giờ có thể được thực hiện). Từ đó việc lợi dụng lỗi tràn bộ nhớ đệm đã tạo ra một cơ chế để kẻ tấn công có thể khéo léo khiến hệ điều hành chuyển điều khiển đến đoạn bố nhớ dữ liệu chứa mã độc nói trên!

**1.2 Tràn bộ đệm trên Heap (Heap overflow).**

Heap là một khu vực của bộ nhớ được sử dụng bởi một ứng dụng và được cấp phát động tại thời gian chạy của các hàm, chẳng hạn như malloc().

Các biến tĩnh được lưu trữ trên stack cùng với dữ liệu được giao sử dụng giao diện malloc.



Nếu một ứng dụng sao chép dữ liệu mà không kiểm tra mặc dù nó phù hợp với nơi được sao chép tới, thì kẻ tấn công có thể cung cấp một lượng lớn dữ liệu cho ứng dụng đó, thực hiện ghi đè lên thông tin quản lý của heap.

Kẻ tấn công làm một bộ nhớ đệm bị tràn lên phần dưới của heap, ghi đè lên các biến động khác gây nên các biến đổi bất ngờ và không mong muốn.

Trong hầu hết các trường hợp, điều này sẽ cho phép kẻ tấn công kiểm soát được việc thực hiện của chương trình.

Ví dụ về heap overflow:

#include<stdlib.h>

#include<string.h>

#include<conio.h>

int main(int argc, char \*\*argv[][]){

char \*input = (char\*) malloc(15);

char \*output = (char\*) malloc(15);

strcpy(output, “normal ouput”);

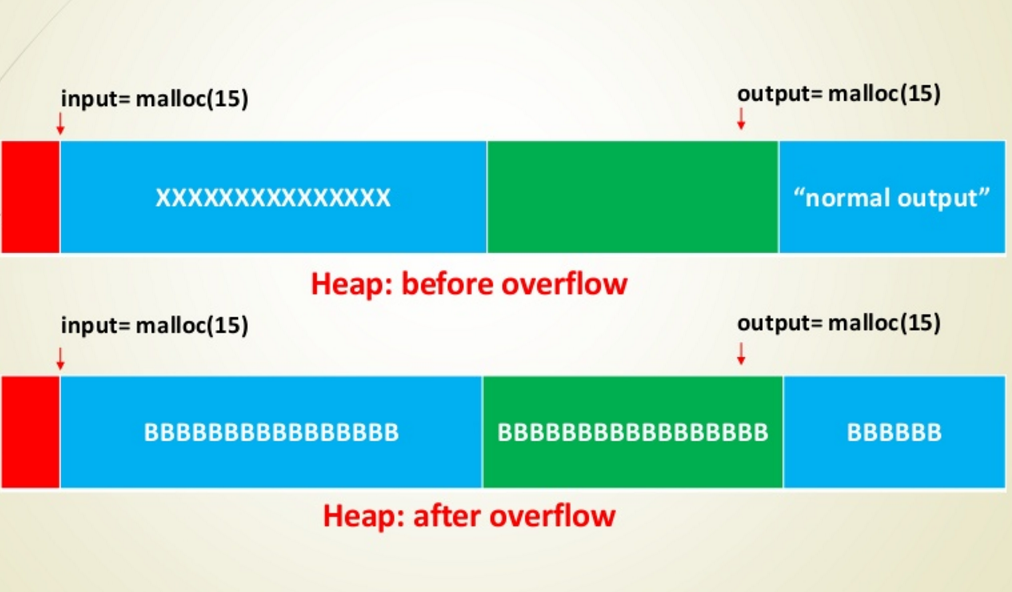
strcpy(input, “BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB”);

printf(“input at %p: %s\n”, input, input);

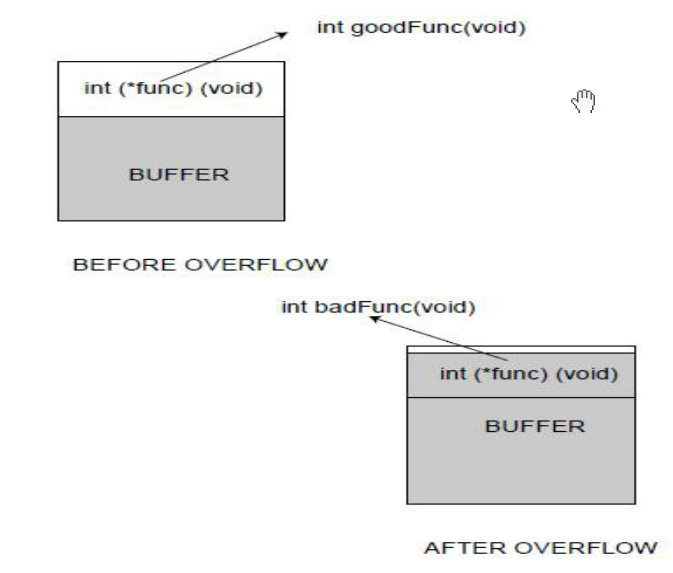
printf(“output at %p: %s\n”, output, output);

return 1;

}



Overwriting poiter (viết lại con trỏ): kẻ tấn công muốn ghi đè lên một con trỏ và làm cho nó trỏ đến những gì họ muốn. Nó có thể trỏ đến một chương trình nào đó.

****

Cuộc tấn công kiểu này cần một số điều kiện tiên quyết trong mã nguồn: một định nghĩa buffer và poiter. Khó khăn của phương pháp này là tìm ra 2 điều kiện tiên quyết như trên. Một khó khăn khác là xác định địa chỉ của vulnerable program.

**1.3 Chống tràn bộ đệm**

### Sử dụng các thư viện an toàn:

Vấn đề tràn bộ đệm thường gặp trong C và C++ vì các ngôn ngữ này để lộ các chi tiết biểu diễn mức thấp của các bộ nhớ đệm với vai trò các chỗ chứa cho các kiểu dữ liệu. Do đó, phải tránh tràn bộ đệm bằng cách gìn giữ tính đúng đắn cao cho các phần mã chương trình thực hiện việc quản lý bộ đệm. Việc sử dụng các thư viện được viết tốt và đã được kiểm thử, dành cho các kiểu dữ liệu trừu tượng mà các thư viện này thực hiện tự động việc quản lý bộ nhớ, trong đó có kiểm tra biên, có thể làm giảm sự xuất hiện và ảnh hưởng của các hiện tượng tràn bộ đệm. Trong các ngôn ngữ này, xâu ký tự và mảng là hai kiểu dữ liệu chính mà tại đó các hiện tượng tràn bộ đệm thường xảy ra; do đó, các thư viện ngăn chặn lỗi tràn bộ đệm tại các kiểu dữ liệu này có thể cung cấp phần chính của sự che chắn cần thiết. Dù vậy, việc sử dụng các thư viện an toàn một cách không đúng có thể dẫn đến tràn bộ đệm và một số lỗ hổng khác; và tất nhiên, một lỗi bất kỳ trong chính thư viện chính nó cũng là một lỗ hổng. Các cài đặt thư viện "an toàn" gồm [The Better String Library](http://bstring.sf.net/), [Arri Buffer API](http://arri.berlios.de/) và [Vstr](http://www.and.org/vstr/). [Thư viện C](https://vi.wikipedia.org/w/index.php?title=Th%C6%B0_vi%E1%BB%87n_C&action=edit&redlink=1) của [hệ điều hành](https://vi.wikipedia.org/wiki/H%E1%BB%87_%C4%91i%E1%BB%81u_h%C3%A0nh) [OpenBSD](https://vi.wikipedia.org/w/index.php?title=OpenBSD&action=edit&redlink=1) cung cấp các hàm hữu ích [strlcpy](https://vi.wikipedia.org/w/index.php?title=Strlcpy&action=edit&redlink=1) [strlcat](https://vi.wikipedia.org/w/index.php?title=Strlcat&action=edit&redlink=1), nhưng các hàm này nhiều hạn chế hơn nhiều so với các cài đặt thư viện an toàn đầy đủ.

### Chống tràn bộ nhớ đệm trên stack:

### Stack-smashing protection là kỹ thuật được dùng để phát hiện các hiện tượng tràn bộ đệm phổ biến nhất. Kỹ thuật này kiểm tra xem [stack](https://vi.wikipedia.org/w/index.php?title=Call_stack&action=edit&redlink=1) đã bị sửa đổi hay chưa khi một hàm trả về. Nếu stack đã bị sửa đổ, chương trình kết thúc bằng một lỗi [segmentation fault](https://vi.wikipedia.org/w/index.php?title=Segmentation_fault&action=edit&redlink=1).

Chế độ [Data Execution Prevention](https://vi.wikipedia.org/w/index.php?title=Data_Execution_Prevention&action=edit&redlink=1) (cấm thực thi dữ liệu) của Microsoft bảo vệ thẳng các con trỏ tới SEH Exception Handler, không cho chúng bị ghi đè.

### Bảo vệ không gian thực thi:

Bảo vệ không gian thực thi là một cách tiếp cận đối với việc chống tràn bộ đệm. Kỹ thuật này ngăn chặn việc thực thi mã tại stack hay heap. Một kẻ tấn công có thể sử dụng tràn bộ đệm để chèn một đoạn mã tùy ý vào bộ nhớ của một chương trình, nhưng với bảo vệ không gian thực thi, mọi cố gắng chạy đoạn mã đó sẽ gây ra một ngoại lệ (*exception*).

Một số [CPU](https://vi.wikipedia.org/wiki/CPU) hỗ trợ một tính năng có tên bit [NX](https://vi.wikipedia.org/w/index.php?title=NX_bit&action=edit&redlink=1) ("No eXecute" - "Không thực thi") hoặc bit [XD](https://vi.wikipedia.org/w/index.php?title=XD_bit&action=edit&redlink=1) ("eXecute Disabled" - "chế độ thực thi đã bị tắt"). Khi kết hợp với phần mềm, các tính năng này có thể được dùng để đánh dấu các [trang dữ liệu](https://vi.wikipedia.org/w/index.php?title=Paging&action=edit&redlink=1) (chẳng hạn các trang chứa stack và heap) là đọc được nhưng không thực thi được.

### Ngẫu nhiên hóa sơ đồ không gian địa chỉ:

Ngẫu nhiên hóa sơ đồ không gian địa chỉ (Address space layout randomization - ASLR) là một tính năng an ninh máy tính có liên quan đến việc sắp xếp vị trí các vùng dữ liệu quan trọng (thường bao gồm nơi chứa mã thực thi và vị trí các thư viện, heap và stack) một cách ngẫu nhiên trong không gian địa chỉ của một tiến trình.

### Kiểm tra sâu đối với gói tin:

Biện pháp kiểm tra sâu đối với gói tin (*deep packet inspection* - DPI) có thể phát hiện các cố gắng từ xa để khai thác lỗi tràn bộ đệm ngay từ biên giới mạng. Các kỹ thuật này có khả năng chặn các gói tin có chứa chữ ký của một vụ tấn công đã biết hoặc chứa một chuỗi dài các lệnh No-Operation (NOP - lệnh rỗng không làm gì), các chuỗi như vậy thường được sử dụng khi vị trí của nội dung quan trọng (*payload*) của tấn công hơi có biến đổi.

Việc rà các gói tin không phải là một phương pháp hiệu quả vì nó chỉ có thể ngăn chặn các tấn công đã biết, và có nhiều cách để mã hóa một lệnh NOP.

# **Định dạng chuỗi (Format String)**

* 1. **Một số khái niệm**

**Format string:**là hành động định dạng đầu ra cho kiểu dữ liệu như integer, float, char …

Ví dụ

#include<stdio.h>

int main(){

char x=’A’;

printf(“%d”,x);

return 0;

}

Đầu ra của chương trình sẽ là “A 65″.

Việc định dạng đầu ra ở đây là in ra kí tự x dưới 2 dạng là char và int. Ta có kết quả như trên là do khi in gặp “%c” chương trình sẽ hiểu là in ra dạng kí tự ASCII ứng với giá trị ‘A’ và in ra “A”, còn khi gặp %d thì chương trình sẽ hiểu là in ra giá trị nguyên của ‘A’ trong bảng ASCII và in ra “65”.

**Format String attack**: là hành động tấn công dựa vào lỗi định dạng chuỗi nó xảy ra khi một dữ liệu dạng chuỗi được nhập vào nhưng được hàm thực thi coi như là một lệnh. Nhờ cách này mà kẻ tấn công có thể truy nhập trái phép vào ngăn xếp và thay đổi dữ liệu trong đó hoặc làm cho chương trình bị treo.

**Format Function:**là chuyển đổi định dạng hàm trong ngôn ngữ C chẳng hạn như printf, fprintf (File Printf).Nhiệm vụ của nó là chuyển đổi những biến trong chương trình ra thành các chuỗi mà chúng ta có thể hiểu được.

**Format String Parameter**: là kiểu định dạng của chuỗi đầu vào nào đó đại diện cho một số hàm chuyển đổi **Format string.** Ví dụ như **%x ,%s**

Sau đây là một số Format Functions:

**Format function** **Miêu tả**

printf In ra chuỗi được định dạng

sprintf In ra một chuỗi

snprintf In ra chuỗi được kiểm tra chiều dài

fprintf Ghi printf vào tệp

Các kiểu định dạng trong C:

|  |  |  |
| --- | --- | --- |
| **Thông số** | **Đầu ra** |  |
| % p | Trỏ tới địa chỉ chứa giá trị của biến |  |
| % d | Số thập phân |  |
| % c | Đầu ra là một kí tự hoặc 1 số |  |
| %u | Số thập phân không dấu |  |
| % x | Dạng số hexa 0-F |  |
| %s | Đầu ra là một chuỗi kí tự |  |
| % n | Đếm độ dài của chuỗi nhập vào |  |

*Một số kiểu đặc biệt:*

**%n**: Đối số được sử dụng như một con trỏ trỏ đến một số nguyên và số lượng kí đầu ra được lưu trữ trong địa chỉ của con trỏ được trỏ đến bởi các đối số trên.

**%s**:Các đối số được sử dụng như một con trỏ trỏ đến một chuỗi mà chuỗi này thay thế cho đầu ra.

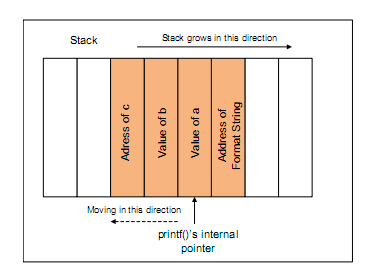
**%x**: Đọc dữ liệu từ stack.

**%a\$n** hay **%a\$x** là để hiển thị giá trị chứa ở vị trí a nào đó

* 1. **Khai thác lỗ hổng**

Giả xử ta có Code như hình dưới.





Nhìn vào Stack chúng ta thấy nó sẽ lấy từ Value a đến Address của c theo lần lượt trong code %d %d %08x.

Vậy còn trường hợp này thì sao, **%d %d %08x** nhưng chỉ có a và b.



Có thể printf () phát hiện sự không tương xứng giữa các kiểu định dạng và các đối số?

* Hàm printf () lấy các đối số từ stack. Nếu chuỗi định dạng cần 3 đối số, nó sẽ lấy ra 3 mục dữ liệu từ ngăn xếp. Trừ khi ngăn xếp được đánh dấu bằng ranh giới, printf () không biết rằng nó đã hết các đối số được cung cấp cho nó.
* Vì không có đánh dấu như vậy printf () sẽ tiếp tục tìm nạp dữ liệu từ ngăn xếp.

Trong một trường hợp không tương xứng, nó sẽ lấy ra một số dữ liệu không thuộc về cuộc gọi hàm này.

Như vậy trong ví dụ trên thì chương trình vẫn sẽ lấy một địa chỉ ở phía dưới giá trị b trong Stack và in ra ngoài.

**2.2.1** **Gây sụp đổ chương trình.**

printf ("%s%s%s%s%s%s%s%s%s%s%s%s");

Đối với mỗi % s, printf () sẽ lấy một số từ ngăn xếp, xử lý số này như một địa chỉ, và in ra các nội dung bộ nhớ chỉ bởi địa chỉ này như là một chuỗi, cho đến khi gặp phải một ký tự NULL (số 0, không phải ký tự 0) .

Vì số được lấy ra bởi printf () có thể không phải là địa chỉ, bộ nhớ được trỏ bởi số có thể không tồn tại (nghĩa là không có bộ nhớ vật lý đã được gán cho địa chỉ đó), chương trình sẽ sụp đổ.

Cũng có thể là con số được lấy ra là một địa chỉ, nhưng là không gian địa chỉ được bảo vệ (Ví dụ như nó được dành riêng cho kernel memory). Trong trường hợp này, chương trình sẽ cũng sụp đổ.

**2.2.2 Xem nội dung Stack.**

printf ("%08x %08x %08x %08x %08x\n");

Ví dụ này cho phép hàm printf lấy ra năm tham số từ ngăn xếp và hiển thị chúng như số hexa có 8 chữ số. Vì vậy, đầu ra có thể có thể giống như sau:

40012980 080628c4 bffff7a4 00000005 08059c04

**2.2.3 Truy cập bộ nhớ bất kỳ.**

Chúng ta phải cung cấp địa chỉ của bộ nhớ. Tuy nhiên, chúng ta không thể thay đổi code; Chúng ta chỉ có thể cung cấp chuỗi định dạng.

- Nếu chúng ta sử dụng printf (%s) mà không chỉ định địa chỉ bộ nhớ, địa chỉ đích cũng sẽ được lấy từ stack bởi hàm printf(). Hàm giữ một con trỏ ngăn xếp ban đầu, do đó, nó biết được vị trí của các tham số trong ngăn xếp.

- Chú ý: chuỗi định dạng thường nằm trên ngăn xếp. Nếu chúng ta có thể mã hóa địa chỉ đích trong chuỗi định dạng, địa chỉ đích sẽ nằm trong ngăn xếp. Trong ví dụ sau, chuỗi định dạng được lưu trữ trong một bộ đệm, nằm trên ngăn xếp.

int main(int argc, char \*argv[])  
{

char user\_input[100];

... ... /\* \*/

scanf("%s", user\_input);

printf(user\_input);

return 0;  
}

Nếu chúng ta có thể buộc printf lấy địa chỉ từ chuỗi định dạng (cũng trên stack), chúng ta có thể kiểm soát địa chỉ.

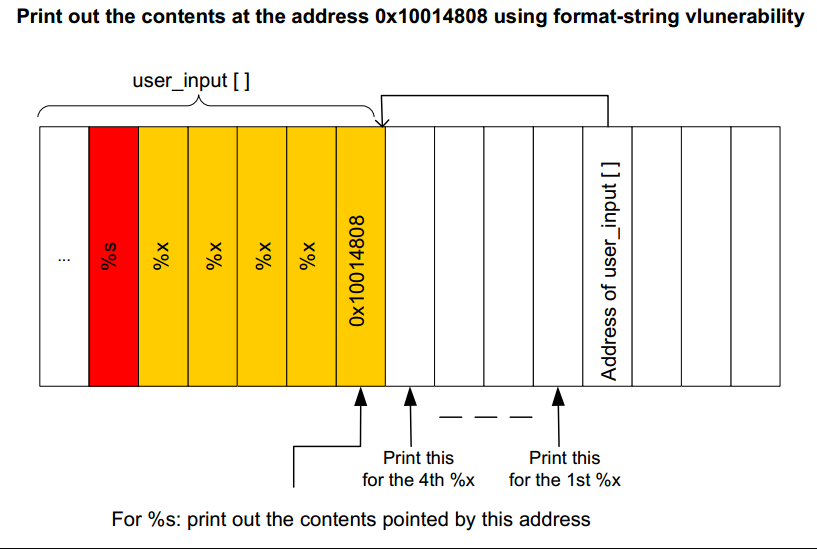
printf ("\x10\x01\x48\x08 %x %x %x %x %s");

\x10 \x01 \x48 \x08 là bốn byte của địa chỉ đích. Trong ngôn ngữ C, \ x10 trong một chuỗi nói với trình dịch để đặt một giá trị hexa 0x10 ở vị trí hiện tại. Giá trị sẽ chiếm một byte duy nhất. Nếu không sử dụng \ x, nếu chúng ta trực tiếp đặt "10" vào một chuỗi, các giá trị ASCII của các ký tự '1' và '0' sẽ được lưu trữ. Giá trị ASCII của chúng là 49 và 48.

% X làm cho con trỏ ngăn xếp di chuyển về phía chuỗi định dạng.

Đây là cách tấn công hoạt động nếu userinput [] có chứa chuỗi định dạng sau:

"\X10\x01\x48\x08 %x %x %x %x %s".

****

In %x thứ 1

In %x thứ 4

% s: in ra các nội dung được chỉ ra bởi địa chỉ này

In ra các nội dung tại 0x10014808 địa chỉ sử dụng lỗ hổng định dạng chuỗi

Về cơ bản, chúng ta sử dụng bốn %x để di chuyển con trỏ printf() về phía địa chỉ chúng ta lưu trữ trong chuỗi định dạng. Khi đến đích, chúng ta dùng %s cho hàm printtf(), khiến cho nó in ra nội dung trong địa chỉ bộ nhớ 0x10014808. Hàm printf () sẽ xử lý nội dung như một chuỗi và in ra chuỗi cho tới khi kết thúc chuỗi (tức là 0).

Không gian ngăn xếp giữa userinput[] và địa chỉ được truyền đến hàm printf () không phải cho printf (). Tuy nhiên, do lỗ hổng định dạng chuỗi trong chương trình, Printf () coi chúng là các đối số để khớp với %x trong chuỗi định dạng.

Khó khăn chính trong cuộc tấn công này là tìm ra khoảng cách user\_input[] và địa chỉ được truyền đến hàm printf (). Khoảng cách này quyết định có bao nhiêu %x cần chèn vào chuỗi định dạng, trước khi đưa ra %s.

**2.2.4 Ghi vào vùng nhớ bất kỳ bằng định dạng %n**

Chuỗi định dạng này in vào vùng nhớ tương ứng tổng số bytes của chuỗi cho đến %n. Ví dụ, chương trình

main () {

int a=0;

printf("1234567%n\n", &a);

printf("a = %d\n", a);

}

cho kết quả là

1234567

a = 7

Cách tiếp cận tương tự như truy cập bộ nhớ bất kỳ , chúng ta có thể dùng printf() để ghi một số nguyên vào bất cứ vị trí nào. Chỉ cần thay thế %s bằng %n trong ví dụ 2.2.3, nội dung ở địa chỉ 0x10014808 sẽ bị ghi đè. Sử dụng tấn công này, những kẻ tấn công có thể thực hiện những việc sau:

-Ghi đè flag quan trọng của chương trình kiểm soát đặc quyền truy cập

- Ghi đè lên địa chỉ trả lại trên ngăn xếp, con trỏ chức năng, vv

**2.3 Cách phòng chống lỗi định dạng chuỗi.**

* Kiểm tra kĩ giá trị input nhập vào
* Sử dụng kĩ thuật Address Random
* Sử dụng các kĩ thuật chống Buffer Overflow
  1. **Lỗi Integer**

**3.1 Một số khái niệm**

Số nguyên được định nghĩa là tập các số {1, 2, 3,…}, các số đối của nó {-1, -2, -3,…} và số 0**.** Trong ngôn ngữ lập trình C (Arduino), ta có thể xem một kiểu dữ liệu số nguyên là một vòng tròn chứa các số trong khả năng biểu diễn của nó, mà trong đó **int** có khả năng biểu diễn các số từ -32,768 đến 32,767 (-215 đến 215-1) (16 bit) và **unsigned int** có khả năng biểu diễn các số từ 0 đến 65535 (0 đến 232 - 1)

Xét đoạn code C sau :

#include<stdio.h>

#include<conio.h>

main() {

unsigned int a=200, b=201 ;

double x, y ;

x = a-b+108.0 ;

y = a-b+108 ;

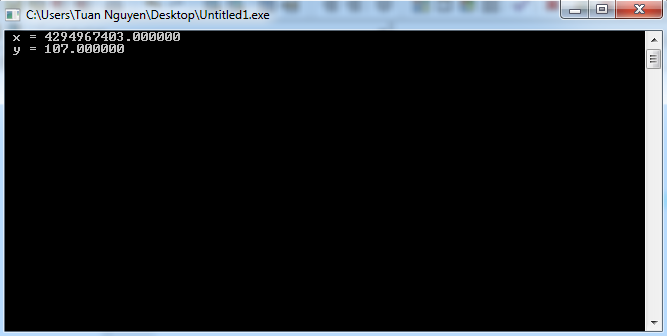
printf(" x = %f\n", x);

printf(" y = %f\n", y);

getch();

}

Kết quả :



Xét lại lỗi đã xảy ra ở đoạn mã nguồn trên, chúng ta thấy rằng, a và b đều là một nguyên không âm . Trong đó, a mang giá trị 200, còn b mang giá trị 201. Nếu xét trên mặt toán học tự nhiên, thì kết quả của biểu thức: a – b + 108.0 = **-1+108**= 107. Số **-1** và **108** ở đâu mà ra? Theo toán học tự nhiên thì 200 – 201 = -1 **⇒** a – b có kết quả là -1**⇒ -1 là kết quả của phép toán a – b**, còn 108.0 có phần thực là 0 **⇒** 108.0 = 108 **⇒** **108 = 108.0**. Trong khi đó, máy tính lại hiểu theo một hướng khác – theo hướng kiểu dữ liệu. Xét lại biểu thức trên, máy tính sẽ thực hiện các phép toán theo thứ tự ưu tiên đúng như trong đại số tự nhiên, như vậy, phép toán a – b sẽ được thực hiện trước, sau đó sẽ lấy kết quả đó cộng với 108.0. Trong lập trình nói chung và ngôn ngữ C nói riêng, có một điểm khác cơ bản với đại số tự nhiên, đó là khái niệm kiểu dữ liệu. Nghĩa là phép toán nào thực hiện trước thì sẽ dùng kiểu dữ liệu có kích thước lớn nhất giữa hai phần tử để lưu trữ kết quả của phép toán đó. Xét lại ví dụ trên, ta thấy a và b đều mang kiểu **unsigned int** **⇒** a – b cũng sẽ mang kiểu **unsigned int** **⇒** a – b không thể nào bằng -1 vì kiểu dữ liệu **unsigned int** không thể hiện giá trị đó!

Tóm lại, khi thực hiện xong phép tính a – b, ta sẽ được giá trị là 232 – 1 do hiện tượng tràn số sau đó chỉ việc lấy số đó cộng thêm với 108.0 ta sẽ được một giá trị như đầu bài với kiểu dữ liệu là **Double**

Vậy ta sẽ có kết quả : 232-1+108.0 = 4294967403

Xét trường hợp 2 a-b+108 cho kết quả đúng 107:

Khi thực hiện xong phép tính a – b, ta sẽ được giá trị là 232 – 1 rồi cộng với 108 kiểu giá trị **int** ta sẽ được kết quả là một số kiểu **unsigned int**

Vậy ta sẽ có kết quả: 232-1+108 = 232+107 do hiện tượng tràn số nên kết quả sẽ là 107.

**3.2 Cách khắc phục và lợi ích việc khắc phục**

Trong tin học, cái cốt lõi vẫn là thực hiện các phép tính của toán học một cách chính xác và nhanh chóng nhất. Vậy với đoạn mã nguồn trên, chúng ta đã không đạt được cái cốt lõi ấy. Như vậy, vấn đề đặt ra là làm thế nào để khắc phục tình trạng tràn số.

Cùng xét lại các thông tin từ ví dụ, ta thấy, có phép trừ trong biểu thức. Theo toán học tự nhiên, khi c = a – b thì c >= 0 nếu a >= b và c < 0 nếu a < b. Trong ví dụ trên, thì a < b ⇒ c < 0 ⇒ tràn số. Như vậy, ta có thể rút ra một luật, khi có phép trừ trong biểu thức số học thì ta không nên dung **unsigned int** (hay bất cứ kiểu dữ liệu nào có chữ **unsigned**) vì như vậy sẽ gây ra tình trạng tràng số. Trong ngôn ngữ pascal, sẽ không có khái niệm tràn số phức tạp như thế này, mà nó chỉ đơn giản là một “lỗi” khi biên dịch!

Ngoài ra, ta cần phải tính toán xem giới hạn tối thiểu và tối đa của biểu thức ta cần tính toán, từ đó lựa ra các kiểu dữ liệu phù hợp.

Việc khắc phục “bệnh” tràn số ở C (Arduino) rất quan trọng, vì điều đó sẽ giúp cho việc lập trình tốn ít thời gian cho việc debug các lỗi liên quan đến cấu trúc dữ liệu, tăng tính tin cậy của thuật toán.

**4. Con trỏ treo (Dangling poiter)**

**4.1 Khái niệm** :

**4.1.1 Dangling poiter**

Trong khi lập trình, chúng ta sử dụng các con trỏ chứa các địa chỉ bộ nhớ của các đối tượng dữ liệu. Con trỏ treo ( Dangling Pointer ) là việc một con trỏ không trỏ tới một vùng nhớ hợp lệ nào. Hiện tượng này xảy ra khi một đối tượng bị deallocate, mà không thay đổi giá trị của con trỏ, khi đó con trỏ vẫn trỏ tới vùng nhớ đã bị deallocate

Bây giờ nếu chúng ta truy cập dữ liệu được lưu trữ ở vị trí bộ nhớ đó bằng cách sử dụng con trỏ lơ lửng thì nó sẽ dẫn đến sự sụp đổ chương trình hoặc hành vi không thể đoán trước.

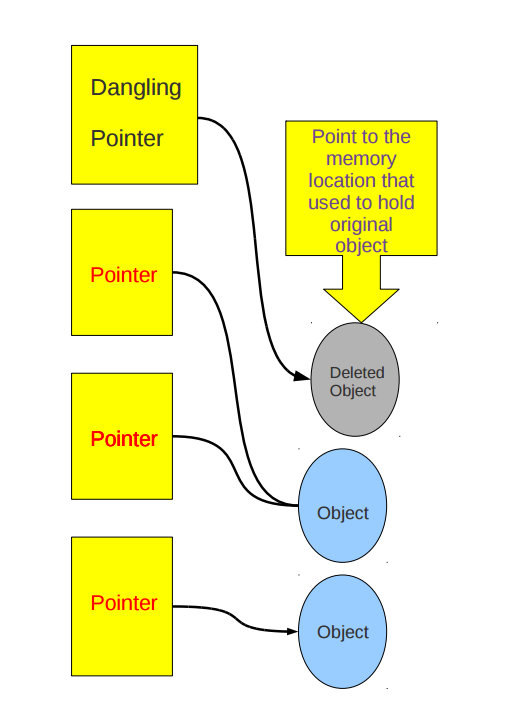
**4.1.2 Vấn đề với Dangling Pointer**

Việc sử dụng các con trỏ lơ lửng có thể dẫn đến một số loại vấn đề khác nhau, bao gồm:

* Hành vi không thể đoán trước được nếu bộ nhớ được truy cập.
* *Lỗi phân đoạn* khi bộ nhớ không còn có thể truy cập.
* Nguy cơ tiềm ẩn về an ninh.

Những loại vấn đề này có thể xảy ra khi:

* Bộ nhớ được truy cập sau khi nó được giải phóng
* Con trỏ được trả về một biến tự động trong một cuộc gọi hàm trước đó



## **4.2 Nguyên nhân của con trỏ tre ( Dangling Pointer ) trong C**

Xét đoạn code sau :

int \*ptr = (int \*)malloc(sizeof(int));  
. . . . .   
. . . . .  
free(ptr); //Hiện tại ptr đang là một dangling pointer

Trong ví dụ trên đầu tiên con trỏ được cấp phát một bộ nhớ và lưu trữ địa chỉ của nó trong ptr. Sau khi thực hiện vài câu lệnh, sẽ giải phóng bộ nhớ. Bây giờ con trỏ ptr đang trỏ đến cùng một địa chỉ bộ nhớ để trở thành con trỏ treo (Dangling Pointer )

**4.3 Giải quyết vấn đề Dangling Pointer**

Rất đơn giản, chỉ cần gán giá trị NULL cho con trỏ sau khi deallocate vùng nhớ mà nó đang trỏ tới. Như là ví dụ sau đây:

int \*ptr = (int \*)malloc(sizeof(int));  
. . . . .   
. . . . .  
free(ptr);  
ptr=NULL //Bây giờ ptr không phải là một dangling pointer nữa