**Bài thực hành 4: Gỡ rối chương trình C++ với GDB**

**TÊN\_BÀI\_LAB: gdb-cpp**

***Bài thực hành này dựa trên lab gdb-cpp trong Labtainer framework của trung tâm Cybersecurity and Cyber Operations, Naval Postgraduate School.***

1. ***Mục đích***

* Bài thực hành này hướng dẫn sử dụng tiện ích GDB để gỡ lỗi một chương trình C++.
* Gỡ lỗi là điều không thể tránh khỏi. Mỗi lập trình viên tại một thời điểm nào đó trong sự nghiệp phải gỡ lỗi một đoạn mã. Có nhiều cách để bắt đầu gỡ lỗi, từ in thông báo ra màn hình, sử dụng trình gỡ lỗi hoặc chỉ nghĩ về những gì chương trình đang làm và đưa ra một phỏng đoán về vấn đề. Trước khi một lỗi có thể được sửa, nguồn của lỗi phải được xác định. Ví dụ, với các lỗi phân đoạn, sẽ rất hữu ích khi biết lỗi seg đang xảy ra trên dòng mã nào. Khi dòng mã được đề cập đã được tìm thấy, sẽ rất hữu ích nếu biết về các giá trị trong phương thức đó, cái gì đã gọi phương thức và tại sao (cụ thể) lại xảy ra lỗi. Sử dụng trình gỡ lỗi làm cho việc tìm kiếm tất cả thông tin này trở nên đơn giản.

1. ***Chuẩn bị thực hành***
   1. *Tìm hiểu lý thuyết*

* Tìm hiểu về gdb
  1. *Tài liệu tham khảo*
  2. *Chuẩn bị môi trường*
* Phần mềm ảo hóa: VMWare Workstation.
* Môi trường Labtainer:
  + Download file ova: <https://nps.box.com/shared/static/2chwo31xgxm2hs4hewp2n4nblroyagwz.ova>
  + Chạy VMWare, rồi: File / Open / LabtainerVM-VMWare.ova
* Khởi động bài lab:
  + Vào terminal, gõ:

*cd ~/labtainer/trunk/scripts/labtainer-student*

*labtainer TÊN\_BÀI\_LAB*

*(chú ý: sinh viên sử dụng email <tên\_tài\_khoản>@stu.ptit.edu.vn của mình để nhập thông tin email người thực hiện bài lab khi có yêu cầu, để sử dụng khi chấm điểm)*

* Dừng bài lab:
  + Vào terminal, tại thư mục:

*~/labtainer/trunk/scripts/labtainer-student*

Gõ:

*stoplab*

* File kết quả bài lab dùng để nộp cùng với file báo cáo nằm trong thư mục:

*/home/student/labtainer\_xfer/TÊN\_BÀI\_LAB*

* Trong trường hợp lỗi máy giữa chừng và cần thực hiện lại bài lab, gõ lệnh:

*labtainer -r TÊN\_BÀI\_LAB*

Nếu có lỗi khác, có thể thử khởi động lại máy ảo và chạy lại.

1. ***Nội dung thực hành***
   1. ***Khởi động lab***
   * Chạy lệnh: *labtainer gdb-cpp* trong terminal của Labtainer
   * Môi trường lab được khởi động. Để giúp minh họa một số nguyên tắc gỡ lỗi, lab sử dụng một ví dụ đang chạy của một chương trình lỗi. Sinh viên sẽ sử dụng gdb để định vị và sửa lỗi trong mã. Mã và một tệp makefile đơn giản nằm trên máy tính gdb-cpp khởi động khi chạy lab này. Mã chương trình này rất đơn giản và bao gồm hai định nghĩa lớp, một nút và một danh sách liên kết. Ngoài ra còn có một chương trình đơn giản để kiểm tra danh sách. Tất cả mã được đặt vào một tệp duy nhất để minh họa quá trình gỡ lỗi dễ dàng hơn. Chương trình và makefile nằm trên máy tính được tạo khi lab khởi động và có thể được nhìn thấy trong thư mục chính.
   1. ***Build và chạy***

Hãy tiếp tục tạo chương trình và chạy chương trình. Chương trình sẽ in ra một số thông báo và sau đó nó sẽ in ra rằng nó đã nhận được tín hiệu lỗi phân đoạn, dẫn đến sự cố chương trình. Với thông tin trên màn hình tại thời điểm này, gần như không thể xác định lý do tại sao chương trình bị lỗi. Hãy bắt đầu gỡ lỗi chương trình này.

* 1. ***Nạp một chương trình trong gdb***

Trước tiên phải khởi chạy trình gỡ lỗi. Trình gỡ lỗi được gọi là gdb và ta có thể cho nó biết tệp nào cần gỡ lỗi tại dấu nhắc shell. Vì vậy, để gỡ lỗi main, ta gõ *gdb main*.

gdb sẽ hiện ra và đợi người dùng gõ lệnh. Tâ cần chạy chương trình để trình gỡ lỗi có thể giúp xem điều gì sẽ xảy ra khi chương trình bị treo. Nhập *run* tại dấu nhắc (gdb). Đây là những gì sẽ thấy khi chạy lệnh:

(gdb) run

Starting program: /home/ubuntu/main

Creating Node, 1 are in existence right now

Creating Node, 2 are in existence right now

Creating Node, 3 are in existence right now

Creating Node, 4 are in existence right now

The fully created list is:

4

3

2

1

Now removing elements:

Creating Node, 5 are in existence right now

Destroying Node, 4 are in existence right now

4

3

2

1

Program received signal SIGSEGV, Segmentation fault.

0x000055555555586c in Node<int>::next (this=0x0) at main.cc:28

28 Node<T>\* next () const { return next\_; }

(gdb)

Chương trình bị lỗi. Vì vậy hãy xem loại thông tin nào có thể thu thập. Kiểm tra sự cố, ta có thể thấy chương trình ở dòng 28 của main.cc, điểm này chỉ đến 0 và chúng ta có thể thấy dòng mã đã được thực thi. Nhưng ta cũng muốn biết cái gì đã gọi phương thức này và muốn có thể kiểm tra các giá trị trong các phương thức gọi. Vì vậy, tại dấu nhắc gdb, nhập *backtrace* cho kết quả sau:

(gdb) backtrace

#0 0x000055555555586c in Node<int>::next (this=0x0) at main.cc:28

#1 0x0000555555555763 in LinkedList<int>::remove (this=0x55555556aeb0,

item\_to\_remove=@0x7fffffffe43c: 1) at main.cc:77

#2 0x00005555555553b1 in main (argc=1, argv=0x7fffffffe558) at main.cc:120

(gdb)

Vì vậy, ngoài những gì đã biết về phương thức hiện tại và các biến cục bộ, bây giờ ta cũng có thể xem những phương thức nào được gọi và tham số của chúng là gì. Ví dụ, có thể thấy rằng chương trình đã được gọi bởi LinkedList <int> :: remove () trong đó mục tham số cần loại bỏ ở địa chỉ 0x7fffffffe43c. Nó có thể giúp hiểu lỗi nếu biết giá trị của mục cần xóa, vì vậy cần xem tiếp giá trị tại địa chỉ của mục cần xóa. Điều này có thể được thực hiện bằng lệnh x sử dụng địa chỉ làm tham số. (”X” có thể được coi là viết tắt của “exam”.) Đây là kết quả của việc chạy lệnh:

(gdb) x 0x7fffffffe43c

0x7fffffffe43c: 0x00000001

(gdb)

Vì vậy, chương trình gặp sự cố trong khi cố gắng chạy LinkedList <int> :: remove với tham số là 1. Bây giờ ta đã thu hẹp vấn đề xuống một hàm cụ thể và một giá trị cụ thể cho tham số.

* 1. ***Các điểm ngắt có điều kiện***

Bây giờ ta biết segfault đang xảy ra ở đâu và khi nào, chúng ta muốn xem chương trình đang làm gì trước khi nó bị treo. Một cách để làm điều này là thực hiện lần lượt từng câu lệnh của chương trình cho đến khi đi đến điểm thực thi mà ta muốn xem điều gì đang xảy ra. Cách làm này đúng, nhưng đôi khi ta có thể muốn chỉ chạy đến một phần mã cụ thể và dừng thực thi tại thời điểm đó để có thể kiểm tra dữ liệu tại vị trí đó. Nếu đã từng sử dụng trình gỡ lỗi, bạn có thể quen thuộc với khái niệm điểm ngắt (breakpoint). Về cơ bản, điểm ngắt là một dòng trong mã nguồn nơi trình gỡ lỗi sẽ ngắt thực thi. Trong ví dụ này, để xem mã trong LinkedList <int> :: remove (), ta đặt một điểm ngắt ở dòng 52 của main.cc.

(gdb) break 52

Breakpoint 1 at 0x29fa0: file main.cc, line 52.

(gdb)

Lúc này Breakpoint 1 được đặt tại main.cc, dòng 52. (Lý do điểm ngắt nhận một số là vì ta có thể tham khảo điểm ngắt sau này, chẳng hạn như nếu muốn xóa nó.) Vì vậy, khi chương trình được chạy, nó sẽ trả lại quyền điều khiển cho trình gỡ lỗi mỗi khi nó đến dòng 52. Điều này có thể không mong muốn nếu phương thức được gọi nhiều lần nhưng chỉ có vấn đề với một số giá trị nhất định được truyền. Các điểm ngắt có điều kiện có thể giúp ta lúc này. Ví dụ: ta biết rằng chương trình bị treo khi LinkedList <int> :: remove () được gọi với giá trị là 1. Vì vậy, ta có thể yêu cầu trình gỡ lỗi chỉ ngắt ở dòng 52 nếu mục cần xóa bằng 1 . Điều này có thể được thực hiện bằng cách ra lệnh sau:

(gdb) condition 1 item\_to\_remove==1

(gdb)

* 1. ***Lệnh Step***

Như vậy ta đã đặt một điểm ngắt có điều kiện và bây giờ muốn xem qua hàm này từng dòng một và xem liệu có thể xác định được nguồn gốc của lỗi hay không. Điều này được thực hiện bằng cách sử dụng lệnh step. gdb có một tính năng hay là khi nhấn enter mà không cần gõ lệnh thì lệnh cuối cùng sẽ tự động được sử dụng. Bằng cách đó, chúng ta có thể thực hiện một cách đơn giản bằng cách nhấn vào phím enter sau khi bước đầu tiên đã được nhập. Đây là kết quả:

(gdb) run

Start it from the beginning? (y or n) y

Starting program: /home/ubuntu/main

Creating Node, 1 are in existence right now

Creating Node, 2 are in existence right now

Creating Node, 3 are in existence right now

Creating Node, 4 are in existence right now

The fully created list is:

4

3

2

1

Now removing elements:

Creating Node, 5 are in existence right now

Destroying Node, 4 are in existence right now

4

3

2

1

Breakpoint 1, LinkedList<int>::remove (this=0x55555556aeb0,

item\_to\_remove=@0x7fffffffe43c: 1)

at main.cc:52

52 Node<T> \*marker = head\_;

(gdb) step

53 Node<T> \*temp = 0; // temp points to one behind as we iterate

(gdb)

55 while (marker != 0) {

(gdb)

56 if (marker->value() == item\_to\_remove) {

(gdb)

Node<int>::value (this=0x7ffff7f1444e <std::ostream::put(char)+94>) at main.cc:30

30 const T& value () const { return value\_; }

(gdb)

LinkedList<int>::remove (this=0x55555556aeb0, item\_to\_remove=@0x7fffffffe43c:

1) at main.cc:75

75 marker = 0; // reset the marker

(gdb)

76 temp = marker;

(gdb)

77 marker = marker->next();

(gdb)

Node<int>::next (this=0x55555556b360) at main.cc:28

28 Node<T>\* next () const { return next\_; }

(gdb)

Program received signal SIGSEGV, Segmentation fault.

0x000055555555586c in Node<int>::next (this=0x0) at main.cc:28

28 Node<T>\* next () const { return next\_; }

(gdb)

Nếu nhìn vào kết quả đầu ra từ việc chạy chương trình, trước hết ta sẽ thấy rằng chương trình chạy mà không bị treo, nhưng có một chỗ rò rỉ bộ nhớ ở đâu đó trong chương trình. (Gợi ý: Nó nằm trong hàm LinkedList <T> :: remove (). Một trong những trường hợp remove không hoạt động bình thường.) Đây là một bài tập cho sinh viên để sử dụng trình gỡ lỗi trong việc định vị và sửa lỗi này .

gdb có thể được thoát bằng cách gõ quit.

1. ***Kết quả cần đạt được***

* Chạy được tất cả các bước như yêu cầu.

1. ***Yêu cầu nộp file kết quả***

* Cần nộp 1 file: trong thư mục: /home/student/labtainer\_xfer/TÊN\_BÀI\_LAB (tên file dạng *email ptit của sinh viên đã nhập.TÊN\_BÀI\_LAB.lab*)