作者: 王駿逸 Simon Wang

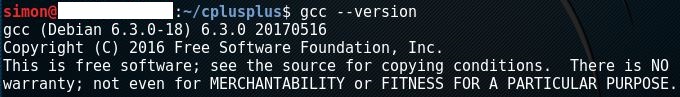
作業環境: Kali Linux 2016 64 bit + GNU gdb (Debian 7.11.1-2) 7.11.1

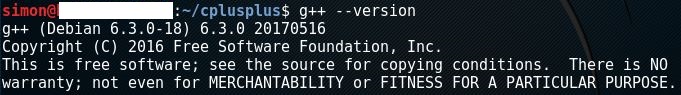
為了要實現徹底理解 Linux Kernel 的運作與可完整有效的對 Linux Driver 進行 Live 除錯，我們必須熟練 QEMU + GDB 的 Kernel Debugging 的建構與操作過程，在建立 QEMU + GDB 的環境中遭遇到了非常多的困難，為了維持住信心、我們不得不暫停 Trial and Error 的過程。

在搜尋針對 QEMU + GDB 的 Kernel Debugging 的錯誤排除的過程中、看到了一些人寫的 QEMU 發問的文章，看到裡面原來即使是沒有 QEMU 之下、 GDB Remote Debugging (Manual) 也是一門很大的學問，為了從目前的障礙中跳脫出來，我們開始有了先摸索清楚 GDB 完整操作的想法。我們從這裡開始一段期間就是以 C++ 練習 + GDB Debugging 的操作來繼續學習，相信再回頭去看 QEMU + GDB 的問題時、可以得到很多意外的答案。

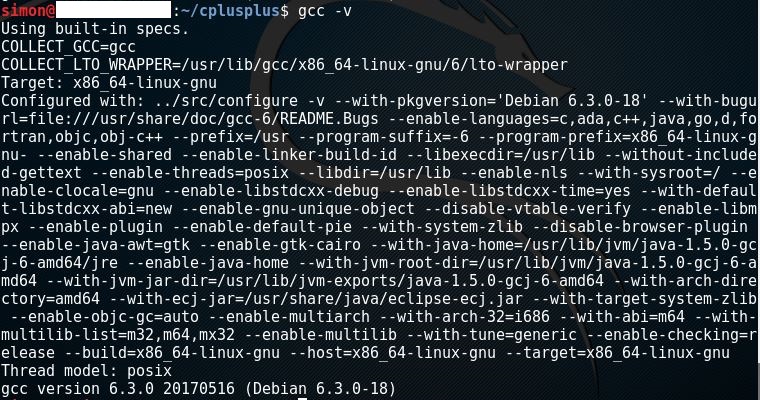
這是屬於我自己的自學學習筆記，內容可能不完全正確與完整、流程可能也不是直觀的一步到位(可能有在錯誤中繞圈來解決的情形)，但是可以提供給初學者一個完整的操作指引、省去很多摸索的步驟，所以還是有其價值。

先來看一下我們的 GCC 的版本:

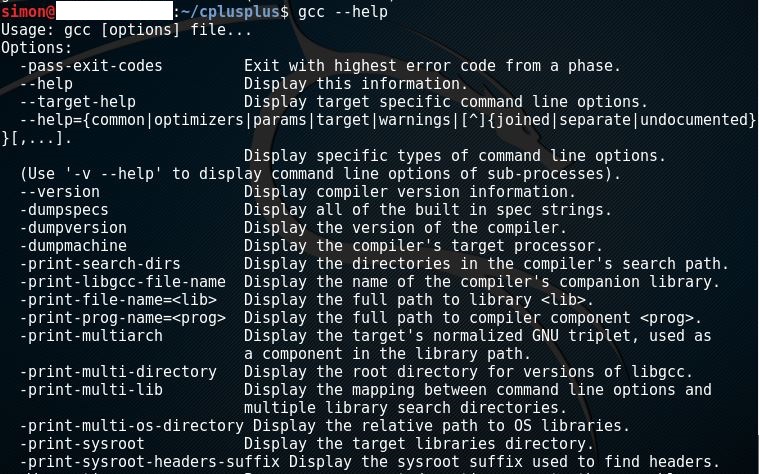


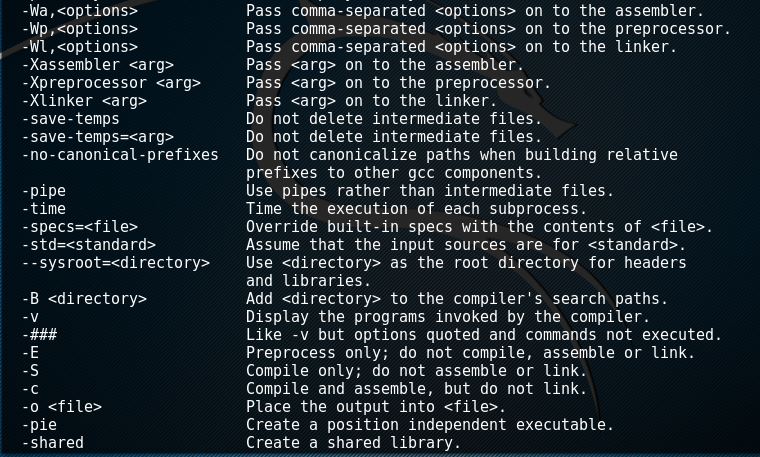


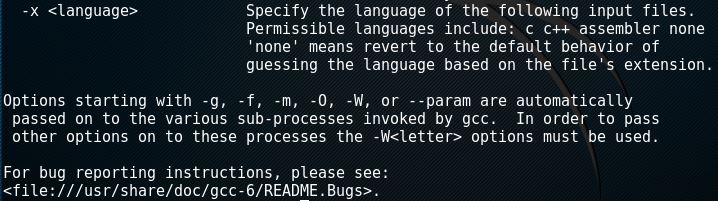
來看看這個 GCC 它更詳細一點的安裝設定選項:



如果我們需要查詢的話， GCC 也有類似的 Help 選項:

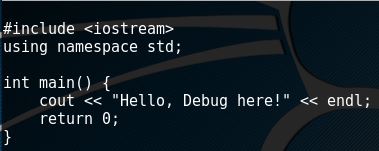






因為這到後面很有參考價值、所以我們把它全部列出來。

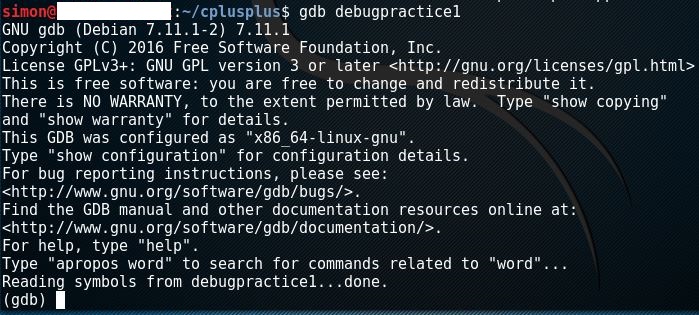
我們的第一個用來 Debug 的測試程式:



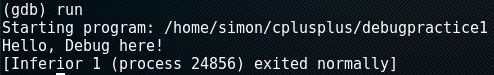
除了一般的用 g++ 來 Compile CPP 程式之外，為了要讓我們的程式帶有 debugging symbols 、我們還要加上 -g 選項:

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GCC Debug 練習6 為了要讓我們的程式帶有 Debugging Symbols 我們還要加上 負g 選項.jpg

開始啟動 GDB :



讓它執行看看:

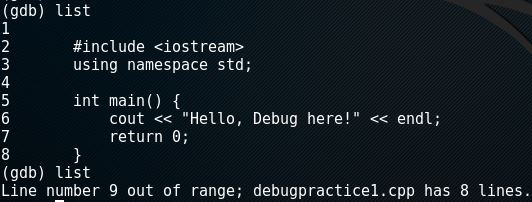


已經是執行完成、 show 出結果了。

若我們要離開 GDB :

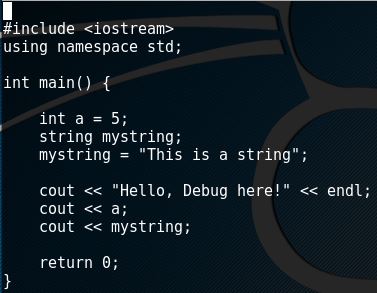
C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GCC Debug 練習9 若我們要離開 GDB.JPG

再次執行 gdb debugpractice1 之後， list 指令可以顯示出 GDB 目前已經執行過的程式碼:



因為我們的程式一瞬間就執行完畢了，所以第一個 list 列出的就是全部的程式碼，第二個 list 想要列印出程式的第 9 行、但已超出範圍、所以就顯示 out of range 訊息。

我們要在 debugpractice1.cpp 中做些修改、加入變數 (variable) :



使用 print 可以顯示程式中變數的值:

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GCC Debug 練習12 使用 list 可顯示程式中變數的值 1.jpg

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GCC Debug 練習13 使用 list 可顯示程式中變數的值 2 都顯示不出來.jpg

為何我們使用了 print 卻不能顯示出 a 跟 mystring 這兩個變數的內容?

經查詢 “gdb: No symbol “i” in current context” 這篇文章之後發現，原因可能是出在我們 Compile 這個 debugpractice1.cpp 程式碼時、所使用的 -o 選項它讓 Compiler optimization 了，所以我們無法在 GDB Debugging 時看到程式碼裡面的變數 (variables) 內容。

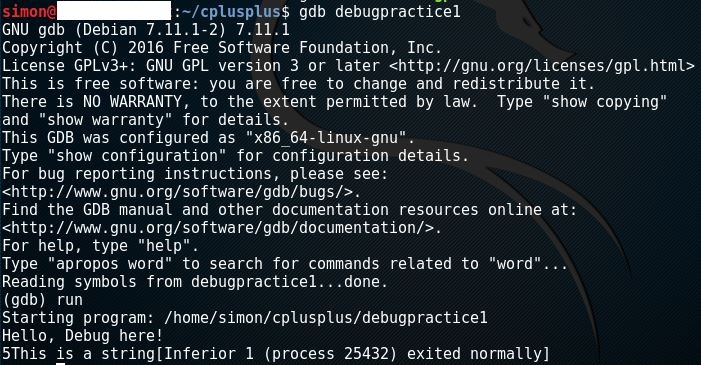
網路上 “Debugging with gdb on a program with no optimization but still there is no symbol in the current context for local variables” 這篇文章有一則解法說:

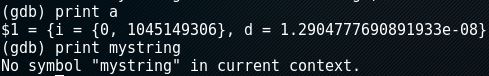
DWARF4 is now the default when generating DWARF debug information. When -g is used on a platform that uses DWARF debugging information, GCC will now default to -gdwarf-4 -fno-debug-types-section. GDB 7.5, Valgrind 3.8.0 and elfutils 0.154 debug information consumers support DWARF4 by default. Before GCC 4.8 the default version used was DWARF2. To make GCC 4.8 generate an older DWARF version use -g together with -gdwarf-2 or -gdwarf-3. The default for Darwin and VxWorks is still -gdwarf-2 -gstrict-dwarf.

所以按照它的解法，若要在 GDB Debugging 中看到 .CPP 程式中的變數內容，我們要加上這樣的 Compile 選項:

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GCC Debug 練習14 使用 list 可顯示程式中變數的值 3 要在 Compile 時 在負g選項後面加上 負gdwarf負2 的選項.jpg

Compile 通過了，再試試看:





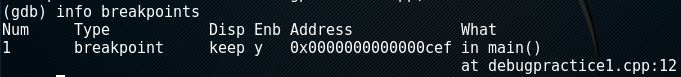
還是一樣顯示不出來。這樣我們開始聯想、是否是因為我們的程式已經執行完畢、所以 GDB 已經完全找不到 a 跟 mystring 這兩個變數中的內容了??

所以我們要先學會 GDB 的中斷指令、讓它停在 a 跟 mystring 這兩個變數剛輸出到螢幕 (cout <<) 的地方:

開始來看中斷點: 若我們要在已知附帶 Debugging Symbols 來 Compile 的程式碼中的某一行設中斷點，則用 break 12 (在第12行下中斷點):

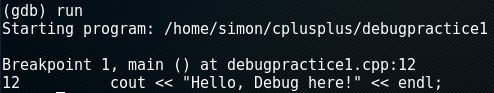
C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GCC Debug 練習19 下中斷點在某一行.jpg

接著就是看目前有設定那些中斷點: info breakpoints

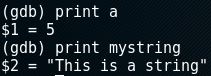


目前只有在我們的程式中設定一個中斷點、所以只有一筆資料，實際後續可能會有多筆。

我們讓它開始執行看看:



現在中斷點是停留在第12行、在指定變數內容之後，我們再一次的來看 a 跟 mystring 這兩個變數現在的內容:



沒錯了，現在可以正確地顯示出這兩個變數的內容了。之前是因為我們的程式已執行完畢、所以當然就撈不到東西。

我們嘗試在 GDB Debugging 中暫時的修改我們的變數內容:

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GCC Debug 練習23 我們嘗試在 Debugging 的過程中暫時的修改我們的變數內容.jpg

會出現這個錯誤原因是出在這一段話:

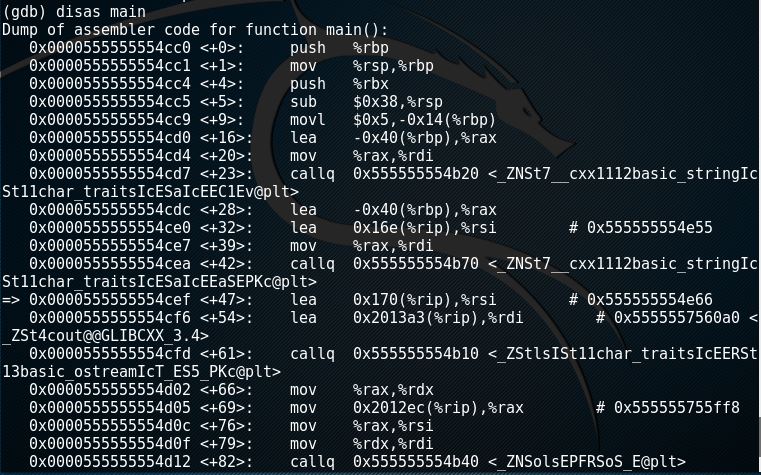
That is, set tries any matching subcommands first, recognizing abbreviations. And since there are multiple commands starting set i (such as set inferior-tty, set input-radix, ...), gdb doesn't know which to choose. So, it complains that the command is ambiguous.

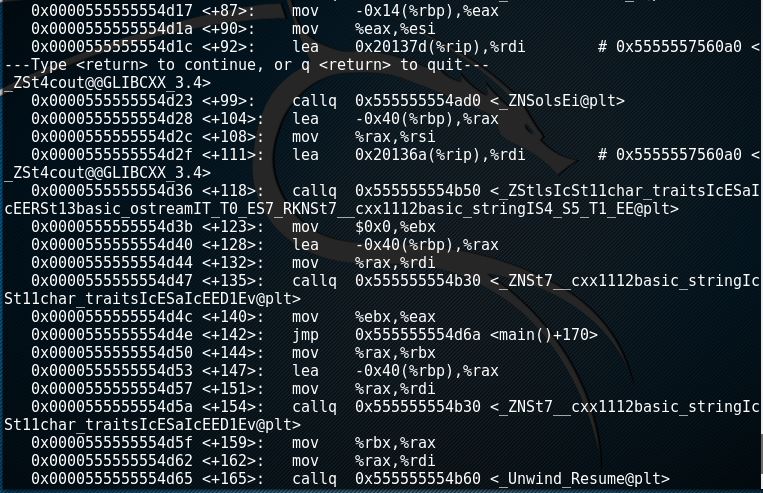
The real command to set a variable is set variable, which is why the final try worked.

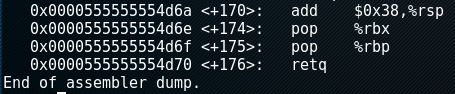
所以現在把它精確的設定為 set variable a = 99 :

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GCC Debug 練習24 我們把它精確的設定為 set variable.jpg

就可以了。再來看一下我們的程式的 Assembly Code 是長怎樣(接下來底層的 Debugging 會需要學習到):







參考 “how can one see content of stack with gdb” 這篇文章，我們可以用 x/32x $esp 的指令來觀察目前被中斷點所暫停的程式中、它的堆疊內容是什麼:

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GCC Debug 練習28 可以用 x 32x esp 的指令來觀察被中斷點暫停的程式中 其堆疊的內容是什麼.jpg

出現了錯誤，經查詢、在64位元的系統下、錯誤原因是出在這裡:

By using $esp, you got the sign-extended 32-bit-truncated value of $rsp, which points to neverland.

Note: GDB provides $sp pseudo-register that is automatically mapped to correct stack pointer register for a given platform.

所以把指令修改為: x/32x $rsp :

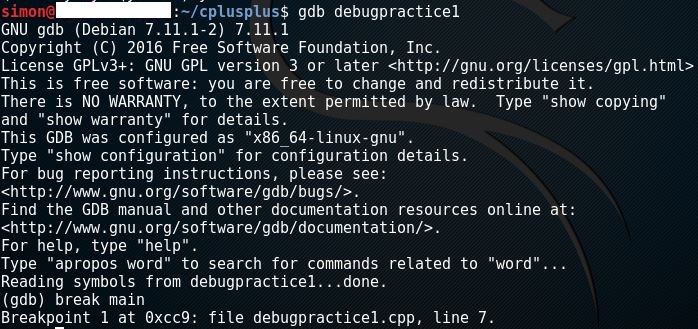


同時:

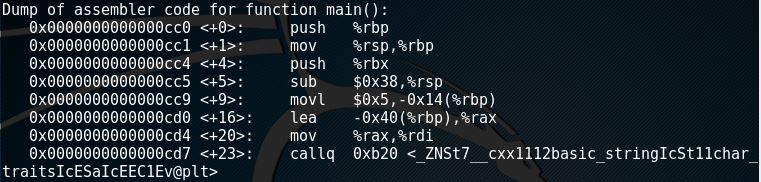


$sp 是虛擬的、但是會呈現出一樣的結果。

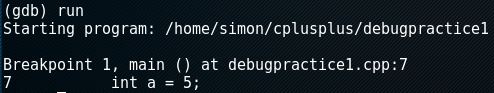
現在我們就重新啟動 gdb debugpractice1 、將中斷點設在 main() 函數上、來練習用 GDB Debugging 追蹤的過程:



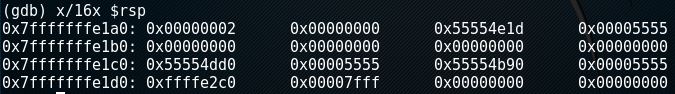
最開頭的部份:



按下 run 讓 GDB 執行:



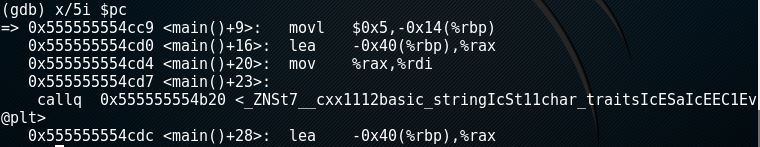
目前的堆疊是這樣:



我們接下來要運用到的指令是 si 單步執行、來看看 GDB 的反應:

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GCC Debug 練習36 把中斷點設在 main 函數上 5 si 單步執行 就到程式碼的下一行了.jpg

在我們執行 gdb debugpractice1 來 Debugging debugpractice1 這個程式、設定中斷點為 main() 函數、執行 run 讓程式跑起來，並執行第一個單步指令 si 之後:



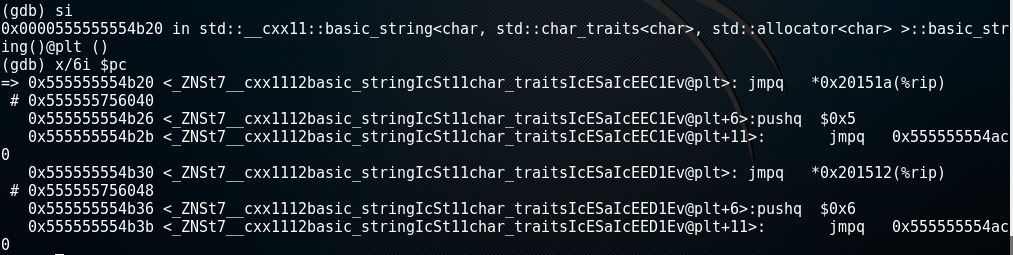
x/5i $pc : 列出目前的執行點 接下來5行的 Assembly 指令。

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\s 是列印出我們的記憶體位置的字串內容 變數 a 的 5 果然已經被放進去了.jpg

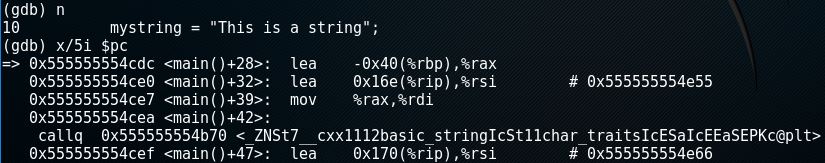
s 是列印出我們的記憶體位置的字串內容，變數 a 的 5 果然已經被放進去了。



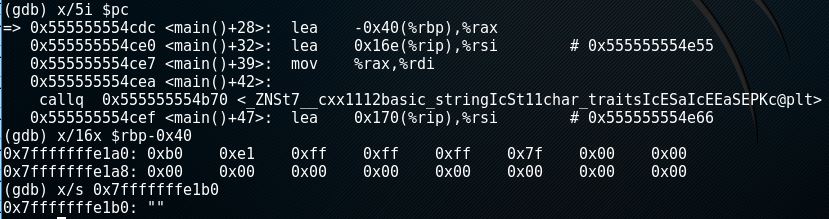
被放進 rdi 後，準備呼叫函數。



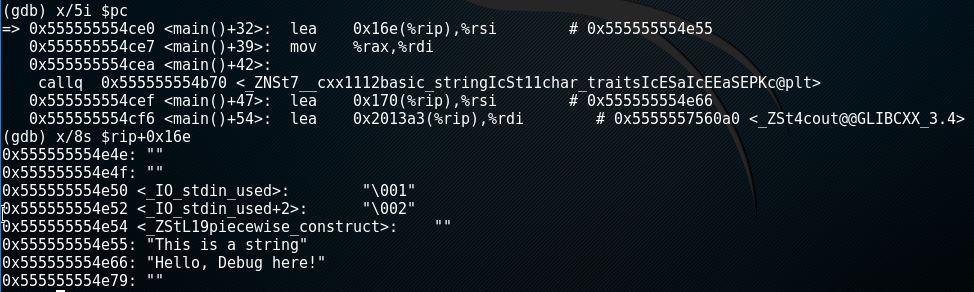
進入了 g++ 的 basic\_stringIcStllchar 的函數。



用 n 指令來直接跳過剛才繁雜的函數。



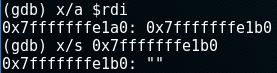
把 雙引號 “ 這個字符所在的記憶體位址，傳給 rax 。



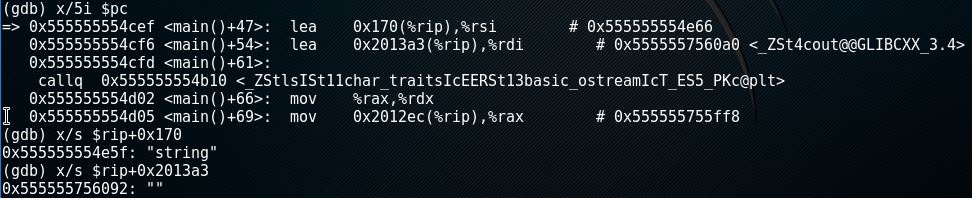
把我們需要用到的字串所存放的 rip 位址，傳給 rsi 之 1。

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\把我們需要用到的字串所存放的 rip 位址 傳給 rsi 2.jpg

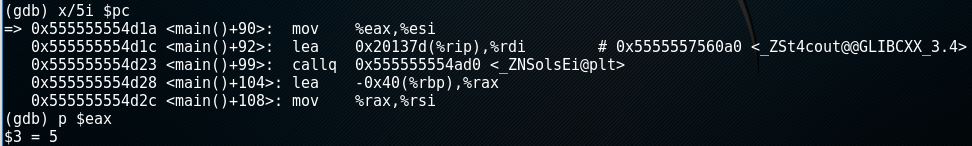
把我們需要用到的字串所存放的 rip 位址，傳給 rsi 之 2。



呼叫 g++ 處理函數前的 rdi 存放著 雙引號 “ 字符的記憶體位址。



把 string 跟 雙引號 “ 這兩個字串，傳送給函數。



把 a 變數的 5 這個值，傳給函數。