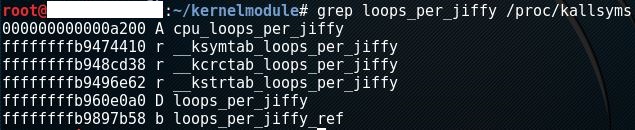
作者: 王駿逸 Simon Wang

作業環境: Kali Linux 2016 64 bit + GNU gdb (Debian 7.11.1-2) 7.11.1 + Linux Kernel Source Code 4.9.13

日期: 2017.8月

現在，我們從使用 GDB 來 “靜態的“ 觀察與 Debug 我們的 Linux Kernel 與 Device Driver 來開始練習起，首先是撈取與觀察在 Kernel Address Space 中的變數:



從 “System.map file and /proc/kallsyms” 、 ”<http://dirac.org/linux/system.map/>” 這兩篇文章我們可以得知:

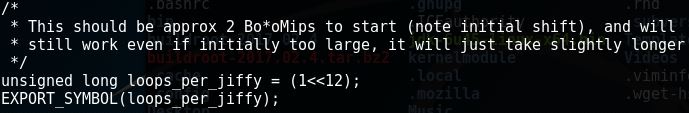
/proc/kallsyms 是代表 /proc 檔案系統格式的動態 Kernel Symbol Table 的 Kernel Data。

System.map 則是代表實際檔案系統中用來儲存目前運作的 Kernel 的 Kernel Symbol Table 的一個檔案。

loops\_per\_jiffy 這個 Kernel 變數，按照 “Linux BogoMIPS” 這篇文章的解釋如下:

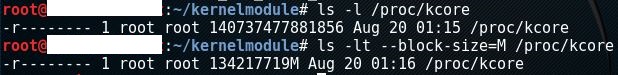
在Linux開機的過程中，通常可以看到這個訊息：Calibrating Delay... xxxx BogoMIPS (lpj=xxxxxxx)，表示Kernel在開機過程會計算CPU在一個jiffy時間內能執行多少次的延時迴圈，這個計算過程是依據CPU的處理速度進行校正，校正結果值會儲存在Kernel變數中，此變數稱為loops\_per\_jiffy，而loops\_per\_jiffy值經過處理後可得到BogoMIPS值。Bogo是Bogus(偽)意思，而MIPS(Million Instructions Per Secon)是每秒百萬指令，所以BogoMIPS就是假的MIPS。透過此網頁<http://www.clifton.nl/bogo-list.html>可知道不同處理器所對應的BogoMIPS。   
  
那為什麼需要loops\_per\_jiffy這個值？因為Kernel Driver有時需要很短且精準的delay (millisecond, microsecond, and nanosecond delays)，此時透過jiffies實現delay是不適用，由於單位不夠小，假設Timer Frequency是250HZ，則一個tick相當於4ms，表示最小delay時間只能到4ms。所以Kernel提供 mdelay(), udelay(), 和 ndelay() 這三個函式來達到極短且精準的delay，而這三個函式是透過loops\_per\_jiffy實現，去計算需要多少個loop operation來得到精確的delay。

同時 loops\_per\_jiffy 在 Linux Kernel Source Code 中它被定義在 init/main.c 裡面:



而另一個重要的 /proc/kcore 則是代表著系統的 physical memory (RAM) ，可以用 GDB 來探索它裡面的 kernel data structures 的狀態，同時這個檔案也有顯示它實際的大小:

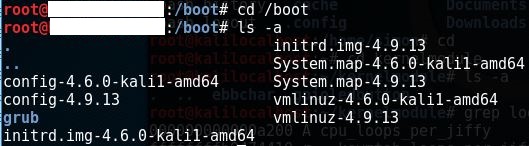
第二條指令則是以 MB 為單位來顯示其檔案的大小:



接下來 GDB 的 Debugging 格式:

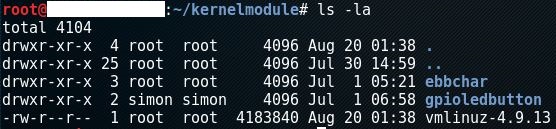
C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Lesson 15 Debugging your kernel and modules with gdb 2.jpg

所以我們來到 /boot 目錄下:

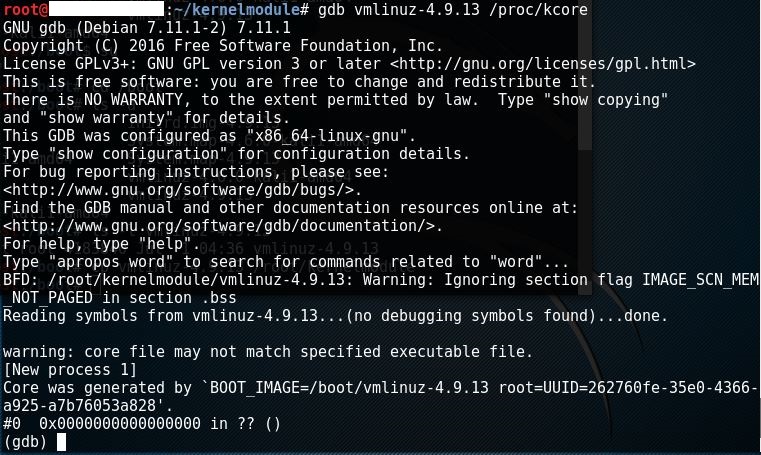


我們目前所 Compiled 且正在執行的 Kernel 執行檔是 vmlinuz-4.9.13 ，為了不破壞到系統運作的起見、我們要 Debug 它時、我們 copy 一份副本來做:

cp vmlinuz-4.9.13 /root/kernelmodule :



按照 GDB Debug 的格式: 第一個參數是 executable file 、第二個參數是 core dump image :

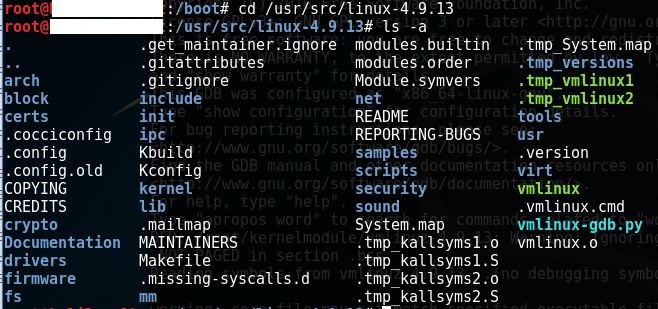


開始來找我們的 loops\_per\_jiffy 的 Kernel 變數:

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Lesson 15 Debugging your kernel and modules with gdb 7 居然找不到 symbol table.jpg

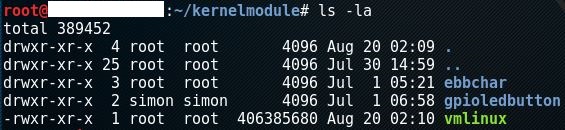
居然找不到 symbol table ?

查詢了一下 “gdb error for 32 bit Ubuntu No symbol table is loaded. Use the "file" command.” 這篇文章的討論串之後發現，我們的問題是在於: vmlinuz-4.9.13 (也就是 vmlinuz 開頭的) 是 compressed kernel image 、裡面是沒有 symbol table 的；而我們既然要用 GDB 來做 Debugging 、我們要用的是 uncompressed kernel image (也就是 vmlinux 開頭的) ， vmlinux 這個檔案的位置是在我們用來 Compile Kernel 的 Kernel Source Code 的主目錄下:

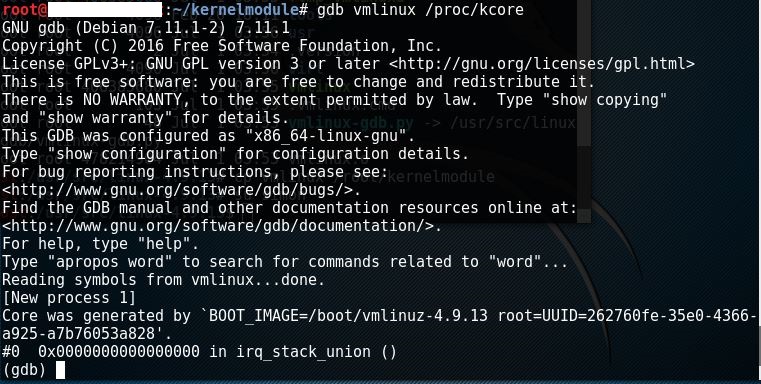


這個綠色的 vmlinux 就是個可執行檔、是我們要的，這也是為什麼我們要先用 Kernel Source Code 來 Compile 一個我們自己的 Kernel 的原因。

我們就把 /usr/src/linux-4.9.13 目錄下的 vmlinux 來 copy 一份來做 Debugging 、避免把它弄壞:

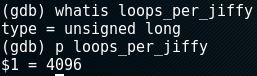


再一次用 gdb vmlinux /proc/kcore :



看到 Reading symbols from vmlinux…done. 這個訊息、就知道現在就有讀取到我們要的 Symbol Table 了。

現在再來看我們原本的 Kernel 變數 loops\_per\_jiffy :



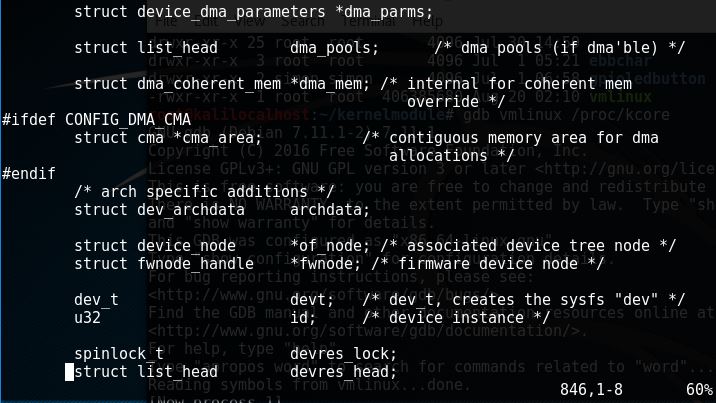
顯示出它是甚麼樣的型別、 p 指令則列印出它的值。

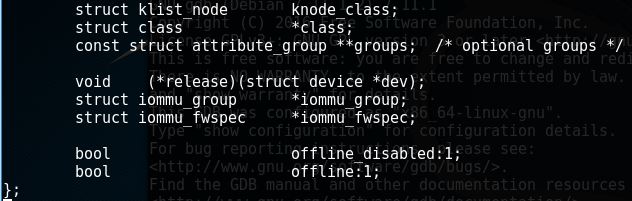
接著我們可以開始探索一些重要的 Kernel Data Structures :

首先我們試試 device 和 device\_driver 這兩個在開發 Linux Device Driver 中常見的結構，透過 <http://elixir.free-electrons.com/linux/v4.9.13/ident/device> 這個網站的查詢，我們得知 device 這個 Kernel Data Structure 是被定義在 include/linux/device.h 中，利用 vi 編輯器中的 / 指令、輸入 struct device 這個關鍵字、並利用 n 指令不斷的向下搜尋之後，我們在 include/linux/device.h 的第711行找到了 device 這個結構的定義:









但是在我們輸入: grep device /proc/kallsyms | more 來一頁一頁的尋找、並沒有找到一個剛好就是代表 struct “device” 的 Symbol ，在 GDB 中:

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Lesson 15 Debugging your kernel and modules with gdb 16 struct device 5.jpg

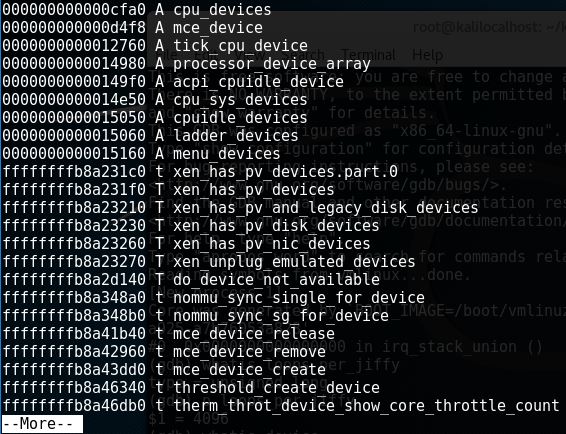
所以就此推斷: struct device 雖然是一個 Kernel Data Structure 、也常在 Device Driver 開發中看到，但它可能不是一個常駐在 Kernel Address Space 中的結構?? 所以在 /proc/kallsyms 中沒有它的 Symbol 。

因此我們再試試看 struct device\_driver :

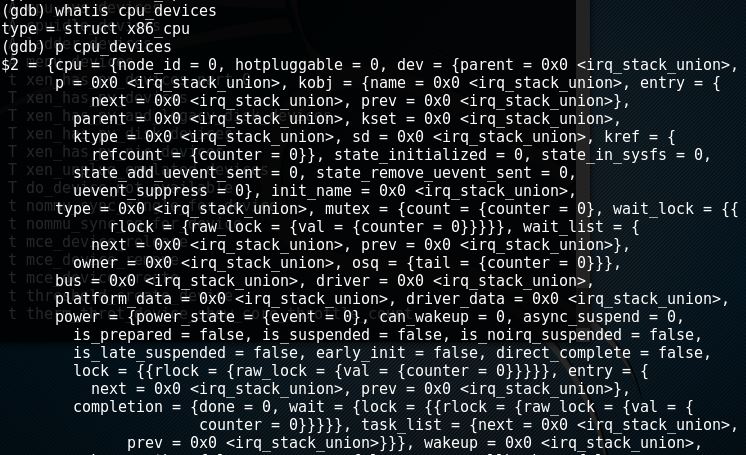


看起來也是同樣的結果、找不到它的 Symbol 。因此，我們現在要找一個已經看到它的 Symbol 存在於 /proc/kallsyms 的 Kernel Data Structure 、好來繼續我們的練習:

再次輸入: grep device /proc/kallsyms | more ，在第一頁顯示我們看到:



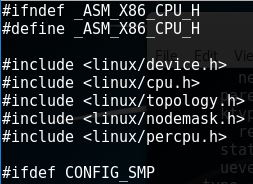
我們就找這個第一個看起來名稱很顯眼的 cpu\_devices 好了，在 GDB 下:



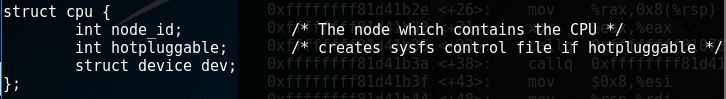
還有很多沒有列印出來……

這個 cpu\_devices 它是屬於 struct x86\_cpu 這個結構，經過 <http://elixir.free-electrons.com/linux/v4.9.13/ident/x86_cpu> 這個網站的查詢，它是被定義在 arch/x86/include/asm/cpu.h 中:

C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Lesson 15 Debugging your kernel and modules with gdb 20 cpu devices 2.jpg



同時裡面的 struct cpu 則是被定義在 include/linux/cpu.h :

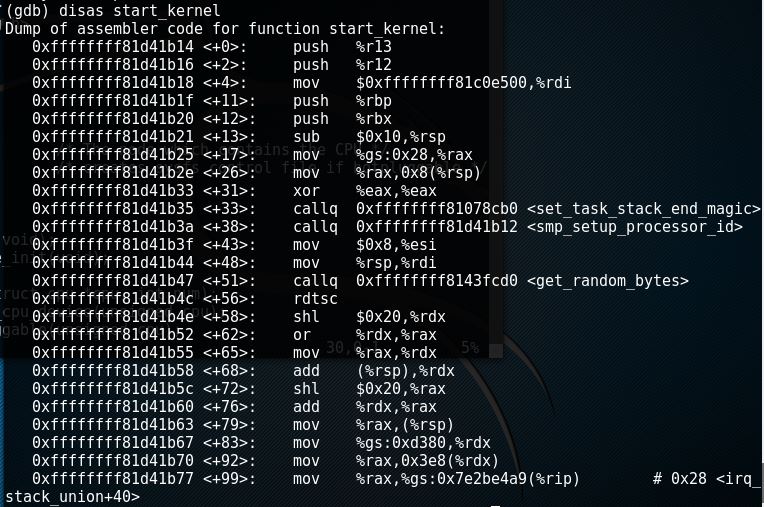


但是因為這個 struct cpu 也不是一個 Symbol ，所以我們要存取它時:

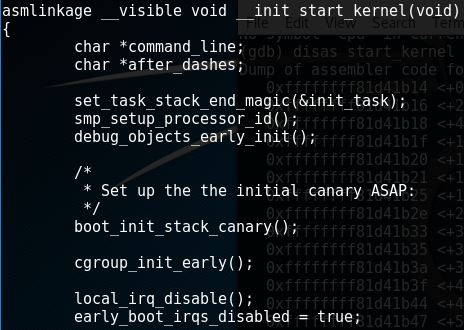
C:\Users\kdbot\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Lesson 15 Debugging your kernel and modules with gdb 23.jpg

就還是沒辦法。

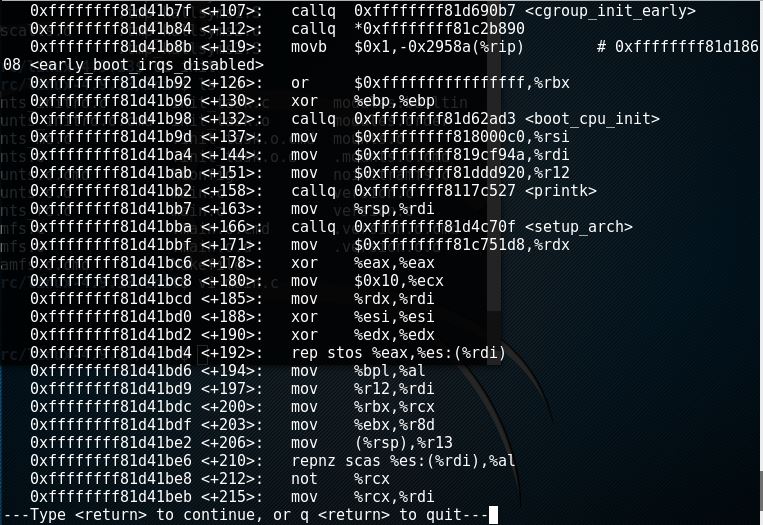
接著，參考到 “Linux內核精析” 這本書， start\_kernel() 函數是 Kernel 啟動執行的起點， 所以我們試著去檢視這個函數:

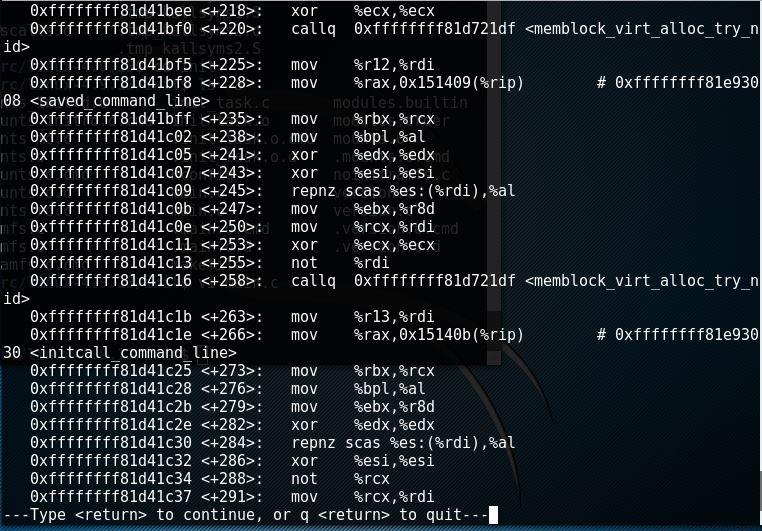


結果果然是有 start\_kernel() 這個函數的 Assembly 執行指令碼，對照著定義在 init/main.c 中的 start\_kernel 函數:



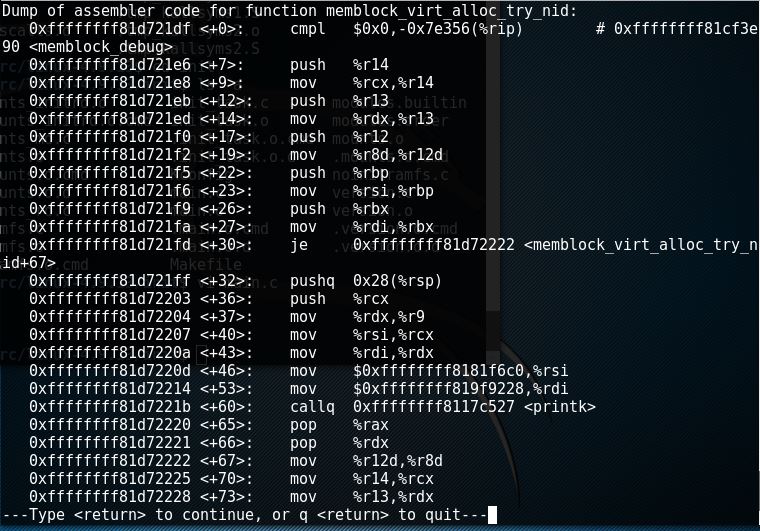
發現了嗎? 我們在上一段的 disas start\_kernel 和 main.c 中 start\_kernel() 函數的程式碼裡面、都有看到 set\_task\_end\_magic 與 smp\_setup\_processor\_id 這兩個函數，代表著我們可以用這種方式來一步一步的觀察追蹤 start\_kernel() 這個 Kernel 中的函數的運作了，這也是我們學習操作 GDB 最終的目的。再往下按 Enter 鍵則跳到繼續下面的程式碼:





當我們閱讀了這些 start\_kernel() 中的 Assembly 指令碼到 callq 0xffffffff81d721df <memblock\_virt\_alloc\_try\_nid> 這一行、想要了解 memblock\_virt\_alloc\_try\_nid() 這個函數的運作時，我們只需要再輸入:

disas memblock\_virt\_alloc\_try\_nid :



這個 memblock\_virt\_alloc\_try\_nid() 的 Assembly 運作指令碼就出現在我們的眼前了。如此，我們就可以算是初步的學會了用 GDB 來做 Linux Kernel 的靜態分析的技巧了。

References:

1. Lesson 15 Debugging your kernel and modules with gdb:

<http://crashcourse.ca/introduction-linux-kernel-programming/lesson-15-debugging-your-kernel-and-modules-gdb>

1. Difference between System.map and /proc/kallsyms:

<https://lists.kernelnewbies.org/pipermail/kernelnewbies/2015-March/013673.html>

1. System.map file and /proc/kallsyms:

<https://stackoverflow.com/questions/28936630/system-map-file-and-proc-kallsyms>

1. Linux BogoMIPS:

<http://reneeciou.blogspot.tw/2013/07/linux-bogomips.html>

1. Linux kernel debugging warning: No Symbol table is loaded:

<https://stackoverflow.com/questions/34505110/linux-kernel-debugging-warning-no-symbol-table-is-loaded>

1. Kernel Symbol Table:

<http://dirac.org/linux/system.map/>

1. 每天一個Linux指令- more (檔案一頁一頁印在螢幕上):

<http://jashliao.pixnet.net/blog/post/159092759-%E6%AF%8F%E5%A4%A9%E4%B8%80%E5%80%8Blinux%E6%8C%87%E4%BB%A4--more-(%E6%AA%94%E6%A1%88%E4%B8%80%E9%A0%81%E4%B8%80%E9%A0%81%E5%8D%B0%E5%9C%A8%E8%9E%A2>

1. 第九章、vim 程式編輯器:

<http://linux.vbird.org/linux_basic/0310vi.php>

1. Linux內核精析:

ISBN 978-7-121-19211-1

郑阿奇 孙承龙

电子工业出版社