

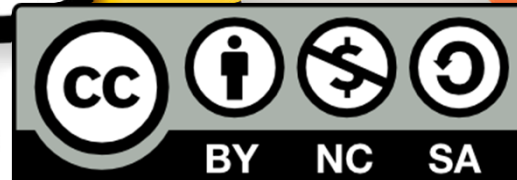


在“hello”背後

中正大學，作業系統實驗室


羅習五 陽春副教授

shiwulo@gmail.com



單元介紹

- 🍏 使用strace瞭解hello用到的system call
- 🍏 實例一：藉由strace分析malloc的動作
- 🍏 實例二：使用strace分析Dropbox
- 🍏 使用ltrace瞭解程式與函數庫的互動



使用strace瞭解hello用到的 system call

hello.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     printf("hello\n");
6.     exit(0);
7. }
```


新的編譯指令

編譯指令

```
$gcc hello.c -o hello
$ls -alh hello
... 8.4K Jan  6 10:09 hello
$gcc hello.c --static -o hello
$ls -alh hello
... 857K Jan  6 10:04 hello
```

--static的含義

- 🍏 我們編譯的程式碼一般而言都會「動態連結」到函數庫（如：libC），--static代表不要連結到函數庫
- 🍏 --static會將該程式（hello）所要用到的程式碼（如：printf），使用copy & paste的方式，將這些程式碼置入hello，因此執行檔案比較大

strace

- 🍏 strace後面接上執行檔案的名稱，strace會執行該執行檔案
- 🍏 strace會攔截該執行檔所發出的所有system call和signal
- 🍏 對於system call而言，strace會列出該system call的名稱，及參數和回傳值
- 🍏 為了確保strace可以「立即」印出這些訊息，strace使用stderr印出訊息
- 🍏 特別注意，以hello為例，strace和hello會將訊息印到同一個console上，不要混淆了

strace hello

這一頁所有的色彩都是，「事後用 powerpoint 加上顏色」

執行strace後面加上執行檔名稱，不要忘了，執行檔前要加上「./」

```
$strace ./hello
execve("./hello", [ "./hello" ], [ /* 62 vars */ ]) = 0
uname({sys="Linux", node="ubuntu", ...}) = 0
brk(0) = 0x78c000
brk(0x78d1c0) = 0x78d1c0
arch_prctl(ARCH_SET_FS, 0x78c880) = 0
readlink("/proc/self/exe", "/home/shiwulo/sp/hello", 4096) = 22
brk(0x7ae1c0) = 0x7ae1c0
brk(0x7af000) = 0x7af000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
getpid() = 1689
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, -1, 0) = 0x7fa7beabb000
write(1, "pid = 1689\n", 11pid = 1689
) = 11
write(1, "hello\n", 6hello
) = 6
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fa7beaba000
read(0,
```

黃色字的hello，是執行檔
hello印出的

那些奇怪的函數在哪裡？

- 🍏 所有Linux的system call可以在底下網址取得名稱及說明
 - 🍀 http://man7.org/linux/man-pages/dir_section_2.html
- 🍏 對Linux system calls進行分類
 - 🍀 <http://www.ibm.com/developerworks/cn/linux/kernel/syscall/part1/appendix.html>
- 🍏 或者直接man該函數
 - 🍀 例如：man 2 brk
 - 🍀 「2」指定Linux內建的「第二本說明書」，第二本說明書詳列所有的system call



實例一： 藉由strace分析malloc的動作

重看hello

```
$strace ./hello
execve("./hello", [ "./hello" ], [ /* 62 vars */ ]) = 0
uname({sys="Linux", node="ubuntu", ...}) = 0
brk(0) = 0x78c000
brk(0x78d1c0) = 0x78d1c0
arch_prctl(ARCH_SET_FS, 0x78c880) = 0
readlink("/proc/self/exe", "/home/shiwulo/sp/hello", 4096) = 22
brk(0x7ae1c0) = 0x7ae1c0
brk(0x7af000) = 0x7af000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
getpid() = 1689
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fa7beabb000
write(1, "pid = 1689\n", 11pid = 1689
) = 11
write(1, "hello\n", 6hello
) = 6
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fa7beaba000
read(0,
```

這一行表示strace執行了hello

這幾行的brk擴增了這個程式的data section

這裡使用mmap擴充了這個程式的data section

觀察記憶體分配情況/proc/pid

```
00400000-004c0000 r-xp 00000000 08:01 1055790  
/home/shiwulo/sp/hello  
006bf000-006c2000 rw-p 000bf000 08:01 1055790  
/home/shiwulo/sp/hello  
006c2000-006c5000 rw-p 00000000 00:00 0  
0078c000-007af000 rw-p 00000000 00:00 0  
7fa7beaba000-7fa7beabc000 rw-p 00000000 00:00 0  
7ffeff05b000-7ffeff07c000 rw-p 00000000 00:00 0  
7ffeff1a2000-7ffeff1a4000 r--p 00000000 00:00 0  
fffffffff600000-fffffffff601000 r-xp 00000000
```

brk增加的data section

mmap增加的data section

vsyscall和vvar的用途為何?

[heap]

[stack]

[vvar]

[vsyscall]

關於proc這個目錄

- 🍏 proc這個目錄是一個虛擬目錄，硬碟上並沒有這個目錄
- 🍏 裡面的所有資料都是動態產生
- 🍏 所有的行程在這個/proc裡面都有相對映的資料夾，該資料夾的名稱是該行程的行程編號（process id）
- 🍏 每個行程的資料夾內，有一個maps的檔案，打開這個檔案就可以看到這個行程的記憶體使用方式

題外話： vsyscall和vvar

- 🍏 放置一些常用的核心變數，例如：gettimeofday、time、getcpu，這些變數放在vsyscall、vvar，因此程式碼存取這些變數不需要進入系統核心，少了 模式切換 (mode change)
- 🍏 vsyscall有安全上的疑慮，目前改為DSO，因為DSO支援ASLR (Address space layout randomization)
- 🍏 ASLR是一種防範記憶體損壞漏洞被利用的電腦保安技術。
- 🍏 Return-to-libc attack是常見的駭客技巧，利用buffer overflow將return address改為libc的函數位址，例如：system()

https://en.wikipedia.org/wiki/Address_space_layout_randomization

VDSO於x86上提供的函數

- 🍏 `__vdso_clock_gettime`
- 🍏 `__vdso_getcpu`
- 🍏 `__vdso_gettimeofday`
- 🍏 `__vdso_time`

malloc是怎樣完成的？

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <sys/types.h>
5.
6.  int main() {
7.      int i;
8.      char* p1;
9.      char* p2;
10.     /*印出行程的pid, 方便我們到/proc目錄裡
        面找到相對映的檔案*/
11.     printf("pid = %d\n", getpid());
12.     printf("malloc(64)\n");
13.     /*配置64byte記憶體*/
14.     p1 = (char*)malloc(64);
15.     printf("p1=%p\n", p1);
16.     printf("malloc 64*4K\n");
17.     /*配置256K記憶體*/
18.     p2 = (char*)malloc(64*4096);
19.     printf("p2=%p\n", p2);
20.     for (i=0; i<64*4096; i++)
21.         p2[i]='0';
22.     /*不要讓程式立即結束, 因為我們還要觀
        察這個程式的記憶體行為*/
23.     while(1);
24. }
```

strace malloc

第一個malloc並未觸發任何system call,
第二個malloc觸發了mmap

```
execve("./a.out", ["/a.out"], [/* 62 vars */]) = 0
uname({sys="Linux", node="ubuntu", ...}) = 0
brk(0) = 0x1eae000
brk(0x1eaf1c0) = 0x1eaf1c0
arch_prctl(ARCH_SET_FS, 0x1eae880) = 0
readlink("/proc/self/exe", "/home/shiwulo/sp/a.out", 4096) = 22
brk(0x1ed01c0) = 0x1ed01c0
brk(0x1ed1000) = 0x1ed1000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
getpid() = 19029
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 27), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fa430af6000
write(1, "pid = 19029\n", 12pid = 19029
) = 12
write(1, "malloc(64)\n", 11malloc(64)
) = 11
write(1, "p1=0x1eb0bc0\n", 13p1=0x1eb0bc0
) = 13
write(1, "malloc 64*4K\n", 13malloc 64*4K
) = 13
mmap(NULL, 266240, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fa430ab5000
write(1, "p2=0x7fa430ab5010\n", 18p2=0x7fa430ab5010
) = 18
```

觀察記憶體分配情況/proc/pid

```
00400000-004c0000 r-xp 00000000 08:01 1048654  
/home/shiwulo/sp/a.out  
006bf000-006c2000 rw-p 000bf000 08:01 1048654  
/home/shiwulo/sp/a.out  
006c2000-006c5000 rw-p 00000000 00:00 0  
01eae000-01ed1000 rw-p 00000000 00:00 0  
[heap]  
7fa430ab5000-7fa430af7000 rw-p 00000000 00:00 0  
7fff0cef0000-7fff0cf1d000 rw-p 00000000 00:00 0  
7fff0cf6e000-7fff0cf70000 r--p 00000000 00:00 0  
7fff0cf70000-7fff0cf72000 r-xp 00000000 00:00 0  
ffffffffffff600000-ffffffffffff601000 r-xp 00000000 00:00 0
```

第一個malloc分配的記憶體
為0x1eb0bc0，落在此處

第二個malloc分配的記憶體
為0x7fa430ab5010，落在此
處

[stack]
[vvar]
[vdso]
[vsyscall]

malloc的行為似乎是...

- 🍏 看起來是比較小的記憶體分配使用brk增加heap
- 🍏 比較大的分配使用mmap分配記憶體
- 🍏 使用malloc2來看一下這個猜測是否正確

malloc2

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <sys/types.h>
5.
6.  int main() {
7.      int i;
8.      char* p;
9.      printf("pid = %d\n", getpid());
10.     printf("malloc 64*4K\n");
11.     /*不斷的跟系統要1B記憶體*/
12.     for (i=0; i<64*4096; i++)
13.         p=(char*)malloc(1);
14.     while(1);
15. }
```

strace malloc2

```
write(1, "malloc 64*4K\n", 13malloc 64*4K
)      = 13
brk(0x10fd000)      = 0x10fd000
brk(0x111e000)      = 0x111e000
brk(0x113f000)      = 0x113f000.
.
.
brk(0x1877000)      = 0x1877000
brk(0x1898000)      = 0x1898000
brk(0x18b9000)      = 0x18b9000
brk(0x18da000)      = 0x18da000
/*看起來我們的猜測是對的*/
```


觀察記憶體分配情況/proc/pid

```
00400000-004c0000 r-xp 00000000 08:01 1048654  
/home/shiwulo/sp/a.out  
006bf000-006c2000 rw-p 000bf000 08:01 1048654  
/home/shiwulo/sp/a.out  
006c2000-006c5000 rw-p 00000000 00:00 0  
010b9000-018da000 rw-p 00000000 00:00 0  
7fd1b1d04000-7fd1b1d05000 rw-p 00000000 00:00 0  
7ffe0a0a2000-7ffe0a0c3000 rw-p 00000000 00:00 0  
7ffe0a11a000-7ffe0a11c000 r--p 00000000 00:00 0  
7ffe0a11c000-7ffe0a11e000 r-xp 00000000 00:00 0  
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0
```

heap大小為8324K，我們跟系統要了256K，看起來libC給我們的比我們實際要的還要多

[heap]

[stack]

[vvar]

[vdso]

[vsyscall]



實例二： 使用strace分析Dropbox

介紹Dropbox

- 🍏 丟到Dropbox的所有檔案都會『自動』上傳到伺服器 (server)
- 🍏 伺服器如果偵測到新的檔案，會自動下載到所有的客戶端 (clients)
- 🍏 最簡單的實現方式
 - 🍀 寫一個程式，不斷的檢查Dropbox相關的目錄是否變動
 - 🍀 優點：簡單
 - 🍀 缺點：沒辦法「立即」查知沒個檔案丟到Dropbox，其次常常檢查會造成CPU使用率過高
- 🍏 使用strace分析Dropbox的行為

看看dropbox的參數

status	get current status of the dropboxd
throttle	set bandwidth limits for Dropbox
help	provide help
puburl	get public url of a file in your dropbox's public
folder	
stop	stop dropboxd
running	return whether dropbox is running
start	start dropboxd
filestatus	get current sync status of one or more files
ls	list directory contents with current sync status
autostart	automatically start dropbox at login
exclude	ignores/excludes a directory from syncing
lansync	enables or disables LAN sync
sharelink	get a shared link for a file in your dropbox
proxy	set proxy settings for Dropbox

strace -c dropbox

% time	seconds	usecs/call	calls	errors	syscall
48.91	0.000067	3	23		mprotect
27.74	0.000038	0	160		mmap
23.36	0.000032	0	826	673	open
0.00	0.000000	0	306		read
0.00	0.000000	0	2		write
...					
0.00	0.000000	0	2		clone
0.00	0.000000	0	1		execve
0.00	0.000000	0	2		wait4
...					

strace似乎沒追蹤到

- 🍏 在Linux上使用dropbox程式，執行後立即結束，看來dropbox不是主要的程式碼
- 🍏 發現dropbox呼叫execv及clone，這表示dropbox呼叫外部程式
- 🍏 使用strace -c -f，將parent和child一網打盡
 - 🍀 strace -c -f dropbox start

發現inotify

```
78.28 19.699726 453 43442 9619 futex
13.58 3.417005 4049 844 1 poll
4.64 1.168000 9733 120 select
2.86 0.720000 720000 1 epoll_wait
...
0.00 0.000000 0 11 epoll_ctl
0.00 0.000000 0 1 inotify_init
0.00 0.000000 0 5 inotify_add_watch
0.00 0.000000 0 35 2 openat
```

...
/*發現一個函數很可疑，inotify因為在unix中檔案都是用inode表示，而notify代表「通知」，因此這個函數很可能是dropbox用來偵測檔案系統變動的函數*/

man inotify

NAME

`inotify_init`, `inotify_init1` - initialize an inotify instance

SYNOPSIS

```
#include <sys/inotify.h>
```

```
int inotify_init(void);  
int inotify_init1(int flags);
```

DESCRIPTION

`inotify_init()` initializes a new inotify instance and returns a file descriptor associated with a new inotify event queue.

*/*用man查一下inotify_init，發現inotify真的是用來偵測檔案系統變動的函數*/*

小結

- 🍏 strace可以用來瞭解自己的程式如何和作業系統核心互動
- 🍏 藉由strace可以瞭解別人的程式如何達到神奇的功能



使用ltrace瞭解程式與函數庫 的互動

malloc3.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <unistd.h>
4.  #include <sys/types.h>
5.
6.  int main() {
7.      int i;
8.      char* p1;
9.      char* p2;
10.     char* p3;
11.     printf("pid = %d\n", getpid());
12.     printf("malloc 256K\n");
13.     p1=(char*)malloc(64*4096);
14.     printf("malloc(1)\n");
15.     p2=(char*)malloc(1);
16.     printf("malloc(1)\n");
17.     p3=(char*)malloc(1);
18. }
```

先使用objdump觀察連結了哪些函數庫

```
shiwulo@vm:~/sp/ch03$ objdump -R a.out
```

```
a.out:      file format elf64-x86-64
```

DYNAMIC RELOCATION RECORDS

OFFSET	TYPE	VALUE
0000000000600ff8	R_X86_64_GLOB_DAT	__gmon_start__
0000000000601018	R_X86_64_JUMP_SLOT	puts@GLIBC_2.2.5
0000000000601020	R_X86_64_JUMP_SLOT	getpid@GLIBC_2.2.5
0000000000601028	R_X86_64_JUMP_SLOT	printf@GLIBC_2.2.5
0000000000601030	R_X86_64_JUMP_SLOT	__libc_start_main@GLIBC_2.2.5
0000000000601038	R_X86_64_JUMP_SLOT	malloc@GLIBC_2.2.5

用ltrace分析malloc3與函數庫的動態行為

```
__libc_start_main(0x40060d, 1, 0x7ffd1b4f6098, 0x400680 <unfinished ...>
getpid() = 20431
printf("pid = %d\n", 20431pid = 20431
) = 12
puts("malloc 256K"malloc 256K
) = 12
malloc(262144) =
0x7fa224d6a010
puts("malloc(1)"malloc(1)
) = 10
malloc(1) = 0x1957010
puts("malloc(1)"malloc(1)
) = 10
malloc(1) = 0x1957030
```

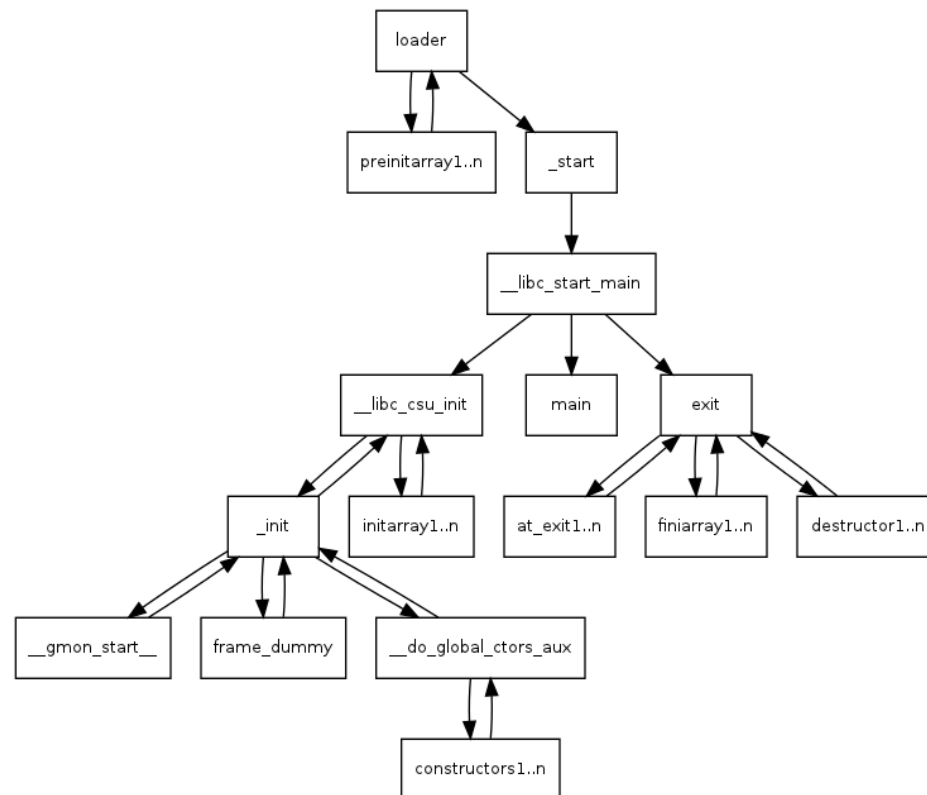
第一個執行的不是main()!!! 好像是__libc_start_main?

- 🍏 會做初始化所有用以呼叫main函數的動作
- 🍏 呼叫main函數
- 🍏 如果main函數回傳 (return) , 控制權會再回到__libc_start_main , 而這個函數會緊接著呼叫exit()
 - 🍀 因此在main中直接回傳 (return) 會間接的執行exit()

使用gdb設定中斷點在“__libc_start_main”

```
(gdb) b __libc_start_main
...
(gdb) bt
#0  __libc_start_main (main=0x4005f6 <main>, argc=1, argv=0x7fffffffdddf8,
    init=0x400670 <__libc_csu_init>, fini=0x4006e0 <__libc_csu_fini>,
    rtdl_fini=0x7ffff7de78e0 <_dl_fini>, stack_end=0x7ffff7d8e8)
    at ../csu/libc-start.c:134
#1  0x00000000400529 in _start ()
/*表示作業系統先呼叫_start*/
```

在main之前



呼叫的函數可能和原始的不同

- 🍏 可以發現，如果是簡單的printf動作，編譯器會使用puts來實現
- 🍏 跟系統要一個byte，但實際上系統給了32byte，這應該是編譯器的最佳化動作

測試 gcc -static malloc3

```
$ gcc -g --static malloc3.c
$ ls a.out -lh
-rwxrwxr-x 1 shiwulo shiwulo 857K Jan 11 11:50 a.out
$ ltrace ./a.out
Couldn't find .dynsym or .dynstr in "/proc/3596/exe"
pid = 3596
malloc 256K
malloc(1)
malloc(1)
```

檔案變得蠻大的

除了a.out以外，ltrace並沒有印出任何東西，因為-static告訴gcc將所有的函數庫複製到a.out中

strip

(捨棄掉所有的symbol, 在嵌入式系統常用)

```
shiwulo@vm:~/sp/ch03$ ls -al a.out
-rwxrwxr-x 1 shiwulo shiwulo 9904 17 09:39 a.out
shiwulo@vm:~/sp/ch03$ strip a.out
shiwulo@vm:~/sp/ch03$ ls -al a.out
-rwxrwxr-x 1 shiwulo shiwulo 6336 17 10:31 a.out
```

小結

- 🍏 介紹了各式各樣的工具分析執行檔案
- 🍏 了解執行檔如何與函數庫進行溝通

作業

- 🍏 統計ls究竟呼叫了哪些system call
- 🍏 拿掉系統中ls的symbol table, 觀察檔案大小的變化
- 🍏 試試看ls是否還可以用
- 🍏 現在還可以使用objdump -R觀察ls連結哪些函數庫嗎