Linux进程信息及内存分布简介

——何良斌

**一、linux进程信息**

本文以thread进程为例，简单创建两个线程。

1、获取进程状态cat /proc/<pid>/status（海思平台和ST平台差不多）

Name: thread

State: S (sleeping)

Tgid: 1199

Pid: 1199

PPid: 1195

TracerPid: 0

Uid: 0 0 0 0

Gid: 0 0 0 0

FDSize: 32

Groups: 0

VmPeak: 13156 kB

VmSize: 13156 kB

VmLck: 0 kB

VmHWM: 256 kB

VmRSS: 256 kB

VmData: 12448 kB

VmStk: 160 kB

VmExe: 544 kB

VmLib: 0 kB

VmPTE: 12 kB

VmSwap: 0 kB

Threads: 3

SigQ: 0/1223

SigPnd: 0000000000000000

ShdPnd: 0000000000000000

SigBlk: 0000000000000000

SigIgn: 0000000000000004

SigCgt: 0000000180000000

CapInh: 0000000000000000

CapPrm: ffffffffffffffff

CapEff: ffffffffffffffff

CapBnd: ffffffffffffffff

Cpus\_allowed: 1

Cpus\_allowed\_list: 0

voluntary\_ctxt\_switches: 5

Name: //进程名字

State: //任务状态, 运行/睡眠/僵死/

Tgid: //线程组号

Pid: //任务ID

PPid: //父进程ID

TracerPid: //接收跟踪该进程信号的进程ID号

FDSize: //文件描述符最大个数

Groups: //启动该进程用户所属的组ID

VmPeak: //进程地址空间的大小,进程运行过程中占用内存的峰值 ，代表占用的最大内存空间

VmSize: //虚拟地址空间大小（total\_vm进程地址空间大小 - reserved\_vm进程在预留的内存间物理页）

VmLck: //已经锁定的物理内存大小（loack\_vm）

VmHWM: //应用程序使用物理内存大小峰值

VmRSS: //应用程序正在使用的物理内存大小

VmData: //程序数据段的大小（所占虚拟内存大小）,存放了初始化数据（total\_vm - shared\_vm - stack\_vm）

VmStk: //任务用户堆栈大小（stack\_vm）

VmExe: //程序拥有的可执行虚拟内存大小，代码段，不包括任务使用的库（end\_code\_start\_code）

VmLib: //被映像到任务的虚拟内存空间的库大小（exec\_lib）

VmPTE: //该进程页表大小

Threads: //线程个数

SigQ: 0/1935 //待处理信号个数

SigPnd: 0000000000000000 //屏蔽位，存储了该线程的待处理信号

ShdPnd: 0000000000000000 //屏蔽位，存储了该线程组的待处理信号

SigBlk: 0000000000000000 //存放被阻塞的信号

SigIgn: 0000000000000001 //存放被忽略信号

SigCgt: 00000001e0001cfe //存放被俘获到的信号

CapInh: 0000000000000000 //能被当前进程执行的程序的继承能力

CapPrm: ffffffffffffffff //进程能够使用的能力

CapEff: ffffffffffffffff //进程有效能力

CapBnd: ffffffffffffffff

Cpus\_allowed: 3

Cpus\_allowed\_list: 0-1

voluntary\_ctxt\_switches: 1125

nonvoluntary\_ctxt\_switches: 301

2、获取进程内存空间映射cat /proc/<pid>/maps**（海思平台）**

反应进程占用的内存区域，每行数据意思：

开始——结束 访问权限 偏移 主设备号:次设备号 i节点 文件

00008000-00090000 r-xp 00000000 00:0f 1014855 /home/thread

00090000-00091000 rw-p 00088000 00:0f 1014855 /home/thread

00091000-00095000 rw-p 00000000 00:00 0

0137d000-0139f000 rw-p 00000000 00:00 0 [heap]

2ab52000-2ab53000 rw-p 00000000 00:00 0

2ab71000-2ab72000 ---p 00000000 00:00 0

2ab72000-2b171000 rw-p 00000000 00:00 0

2b218000-2b219000 ---p 00000000 00:00 0

2b219000-2b818000 rw-p 00000000 00:00 0

7e973000-7e99a000 rw-p 00000000 00:00 0 [stack]

ffff0000-ffff1000 r-xp 00000000 00:00 0 [vectors]

3、获取进程内存空间映射cat /proc/<pid>/maps**（ST平台）**

00008000-00078000 r-xp 00000000 00:13 1047568 /home/n7/thread

0007f000-00081000 rw-p 0006f000 00:13 1047568 /home/n7/thread

00081000-00084000 rw-p 00000000 00:00 0

01499000-014bb000 rw-p 00000000 00:00 0 [heap]

75fe4000-75fe5000 ---p 00000000 00:00 0

75fe5000-767e4000 rw-p 00000000 00:00 0 [stack:437]

767e4000-767e5000 ---p 00000000 00:00 0

767e5000-76fe5000 rw-p 00000000 00:00 0 [stack:436]

7e9f7000-7ea1e000 rw-p 00000000 00:00 0 [stack]

ffff0000-ffff1000 r-xp 00000000 00:00 0 [vectors]

4、进程maps解析

从maps可以得到进程的一个内存布局图，不同颜色代表程序不同段映射：



（1）代码段

00008000-00090000 r-xp 00000000 00:0f 1014855 /home/thread

（2）数据和BSS段

00090000-00091000 rw-p 00088000 00:0f 1014855 /home/thread

00091000-00095000 rw-p 00000000 00:00 0

数据和bss段与代码段基本都是紧挨着的。

（3）堆heap

01499000-014bb000 rw-p 00000000 00:00 0 [heap]

向上增长

（4）mmap区域

thread进程主要是创建了两个线程，每个线程的堆栈大小为6M。

2ab71000-2ab72000 ---p 00000000 00:00 0 //4K

2ab72000-2b171000 rw-p 00000000 00:00 0 //6M

2b218000-2b219000 ---p 00000000 00:00 0 //4K

2b219000-2b818000 rw-p 00000000 00:00 0 //6M

1. 海思与ST平台对比发现创建线程时mmap增长方向刚好相反，见第四节。
2. 线程通过mmap方式映射，为啥每个线程都多一个4K，见第三节。

（5）stack区域

7e973000-7e99a000 rw-p 00000000 00:00 0 [stack]

位于地址高位，向下增长。

**二、进程镜像信息与maps关系**

使用objdump工具可以查找thread二进制文件的地址和符号信息，和maps内容对比发现基本是一致的。

1、代码段

arm-hisiv200-linux-objdump -t thread | grep "\.text"得到如下信息：

……

0004e140 g F .text 00000008 \_\_wmemcpy

0001b810 g F .text 00000008 \_IO\_iter\_next

0006e34c g F .text 00000cb0 \_dl\_close\_worker

00022cd0 g F .text 00000344 \_\_valloc

0000929c g F .text 00000198 \_\_pthread\_init\_static\_tls

00055624 g F .text 00000014 \_\_geteuid

00026f68 g F .text 0000013c \_wordcopy\_bwd\_aligned

…….

从第一列地址看，代码段的地址范围与上面的maps获取到的地址范围是一致的。

2、数据段：

arm-hisiv200-linux-objdump -t thread | grep "\.data"得到如下信息：

00090024 l d .data.rel.ro 00000000 .data.rel.ro

000900c8 l d .data 00000000 .data

000900d0 l O .data 00000008 stack\_used

……………..

0009056c g O .data 00000004 \_dl\_make\_stack\_executable\_hook

00090568 g O .data 00000004 \_dl\_correct\_cache\_id

00090538 w O .data 00000004 \_\_memalign\_hook

00090240 g O .data 000000a0 \_IO\_2\_1\_stderr\_

00090550 g O .data 00000004 \_\_progname\_full

数据地址与maps一致

3、bss段

arm-hisiv200-linux-objdump -t thread | grep "\.bss"得到如下信息

000907d0 l d .bss 00000000 .bss

000907d0 l O .bss 00000001 completed.5624

000907d4 l O .bss 00000004 in\_flight\_stack

000907d8 l O .bss 00000004 stack\_cache\_actsize

000907e4 l O .bss 00000004 stack\_cache\_lock

000907e8 l O .bss 00000008 \_\_nptl\_threads\_events

000907f0 l O .bss 00000004 \_\_nptl\_last\_event

000907f8 l O .bss 00000001 \_\_nptl\_initial\_report\_events

00092800 l O .bss 00000208 static\_slotinfo

00092a08 l O .bss 00000200 static\_dtv

…..

000941d4 g O .bss 00000004 \_dl\_osversion

000941d8 g O .bss 00000004 \_dl\_inhibit\_rpath

000941dc g O .bss 00000004 \_dl\_pagesize

000941e0 g O .bss 0000004c \_dl\_ns

地址同maps对比发现，最开始的地方使用了数据段的地址空间。

**三、strace跟踪进程**

通过strace工具跟踪thread创建两个线程的过程：

/home/n5x/strace home/n5x/thread 2：

execve("home/n5x/thread", ["home/n5x/thread", "2"], [/\* 8 vars \*/]) = 0

uname({sys="Linux", node="host", ...}) = 0

brk(0) = 0x163e000

brk(0x163ed04) = 0x163ed04

set\_tls(0x163e4c0, 0x90014, 0xffffffe0, 0x14, 0x92800) = 0

set\_tid\_address(0x163e068) = 1199

set\_robust\_list(0x163e070, 0xc) = 0

futex(0x7e85cd14, FUTEX\_WAKE\_PRIVATE, 1) = 0

futex(0x7e85cd14, FUTEX\_WAIT\_BITSET\_PRIVATE|FUTEX\_CLOCK\_REALTIME, 1, NULL, 163e08c) = -1 EAGAIN (Resource temporarily unavailable)

rt\_sigaction(SIGRTMIN, {0xd870, [], SA\_SIGINFO|0x4000000}, NULL, 8) = 0

rt\_sigaction(SIGRT\_1, {0xd714, [], SA\_RESTART|SA\_SIGINFO|0x4000000}, NULL, 8) = 0

rt\_sigprocmask(SIG\_UNBLOCK, [RTMIN RT\_1], NULL, 8) = 0

getrlimit(RLIMIT\_STACK, {rlim\_cur=6144\*1024, rlim\_max=6144\*1024}) = 0

brk(0x165fd04) = 0x165fd04

brk(0x1660000) = 0x1660000

fstat64(1, {st\_mode=S\_IFCHR|0600, st\_rdev=makedev(204, 64), ...}) = 0

ioctl(1, SNDCTL\_TMR\_TIMEBASE or SNDRV\_TIMER\_IOCTL\_NEXT\_DEVICE or TCGETS, {B115200 opost isig icanon echo ...}) = 0

mmap2(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x2ab02000

write(1, "stacksize 6291456\n", 18stacksize 6291456

) = 18

mmap2(NULL, 6291456, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x2ab5a000

mprotect(0x2ab5a000, 4096, PROT\_NONE) = 0

clone(child\_stack=0x2b158e38, flags=CLONE\_VM|CLONE\_FS|CLONE\_FILES|CLONE\_SIGHAND|CLONE\_THREAD|CLONE\_SYSVSEM|CLONE\_SETTLS|CLONE\_PARENT\_SETTID|CLONE\_CHILD\_CLEARTID, parent\_tidptr=0x2b159368, tls=0x2b1597c0, child\_tidptr=0x2b159368) = 1200

write(1, "stacksize 6291456\n", 18stacksize 6291456

) = 18

mmap2(NULL, 6291456, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x2b166000

mprotect(0x2b166000, 4096, PROT\_NONE) = 0

clone(child\_stack=0x2b764e38, flags=CLONE\_VM|CLONE\_FS|CLONE\_FILES|CLONE\_SIGHAND|CLONE\_THREAD|CLONE\_SYSVSEM|CLONE\_SETTLS|CLONE\_PARENT\_SETTID|CLONE\_CHILD\_CLEARTID, parent\_tidptr=0x2b765368, tls=0x2b7657c0, child\_tidptr=0x2b765368) = 1201

write(1, "i=2\n", 4i=2

) = 4

write(1, "wait a moment, \n", 16wait a moment,

) = 16

rt\_sigprocmask(SIG\_BLOCK, [CHLD], [], 8) = 0

rt\_sigaction(SIGCHLD, NULL, {SIG\_DFL, [], 0}, 8) = 0

上面的蓝色和紫色分别创建了一个线程，其中有一个mprotect函数创建了4KB与mmap对应的内存保护区，该区域用户无法操作，如果意外操作就会出错。这就是为什么在maps中每个线程会有一个4KB的原因。

**四、不同内核进程内存布局**

对比海思（3.0.8）和ST（3.4.7）平台，maps中线程mmap增长方向是相反的。说明不同的内核版本采用不同的内存布局，对于早期的内核采用3.0.8的经典内存布局，最新的都是3.4.7的灵活内存布局。

1、经典内存布局

经典布局mmap的增长方向：向上增长， 如图：



对于经典模式，heap和mmap都是向上增长的，于是在内核中就为heap保留了一定空间：heap ~mmap起始地址。

mmap起始地址计算方法：

在内核代码/arch/arm/mm/mmap.c

start\_addr = TASK\_UNMAPPED\_BASE

#define TASK\_UNMAPPED\_BASE (UL(CONFIG\_PAGE\_OFFSET) / 3)

CONFIG\_PAGE\_OFFSET = 0xc0000000（对于2G的就是0x80000000）

因此，在这种情况下，mmap可用区域并不是3G，而是2\*( CONFIG\_PAGE\_OFFSET) /3。

2、灵活内存布局

灵活内存布局的增长方向：向下增长，如图：



从图中可以看出，栈至顶向下扩展，并且栈是有界的。对至底向上增长，mmap映射区域至顶向下扩展，直至耗尽虚拟地址空间的剩余区域。

当栈中压入数据超出其容量就会耗尽栈所对应的内存区域，这将触发一个页故障（page fault），并被linux的expand\_stack()处理，调用acct\_stack\_growth来检查是否有合适的地方用于栈的增长。如果栈的大小低于RLIMIT\_STACK，那么一般情况下栈会加长。

另外，可以看出栈和mmap映射区域不是从一个固定的地址开始，而是程序启动时有一个随机的偏移，这样使得使用缓冲区溢出攻击的方法更加困难。如果想使地址固定，设置

/proc/sys/kernel/randomize\_va\_space 为0。

也可以将灵活内存布局设置为经典内存布局，设置变量/proc/sys/vm /legacy\_va\_layout为1。

**五、其他补充**



