

第17讲 Linux进程的地址空间

背景回顾：Linux 从一个初始的程序 (状态机) 开始，构建出了整个应用程序世界——通过 fork, execve, exit，我们可以在操作系统中创建出很多并发/并行执行的程序。

本讲内容：在我们的状态机模型中，进程的状态由 (M,R) 两部分组成；其中 R (register) 是由体系结构决定的，而 M (memory) 则还有一些未解开的谜题：程序在初始时，并不是所有的内存都可以访问的，但我们却又的确可以申请很大的内存。这是如何实现的？

- 进程的地址空间
- mmap 系统调用
- 三类游戏外挂的实现原理
 - 金山游侠：内存修改
 - 按键精灵：GUI 事件发送
 - 变速齿轮：代码注入

1. Linux进程的地址空间

Linux 进程的地址空间

两个很基本 (但也很困难) 的问题

- 以下程序的 (可能) 输出是什么？

```
printf("%p\n", main);
```

- 何种指针访问不会引发 segmentation fault？

```
char *p = random();  
*p; // 什么时候访问合法？
```

这种小问题问 GPT 就好了

查看进程的地址空间

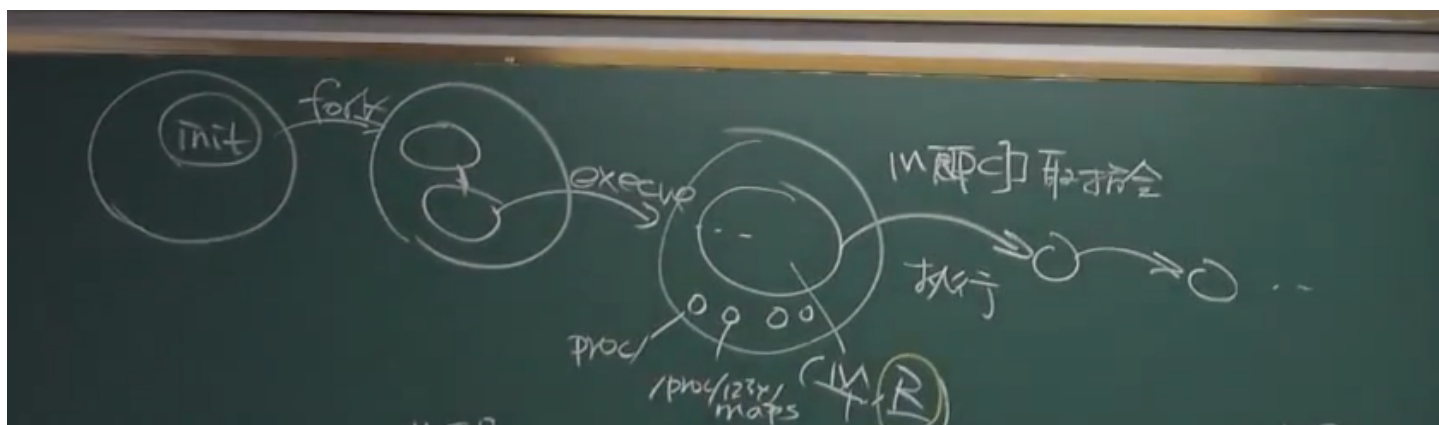
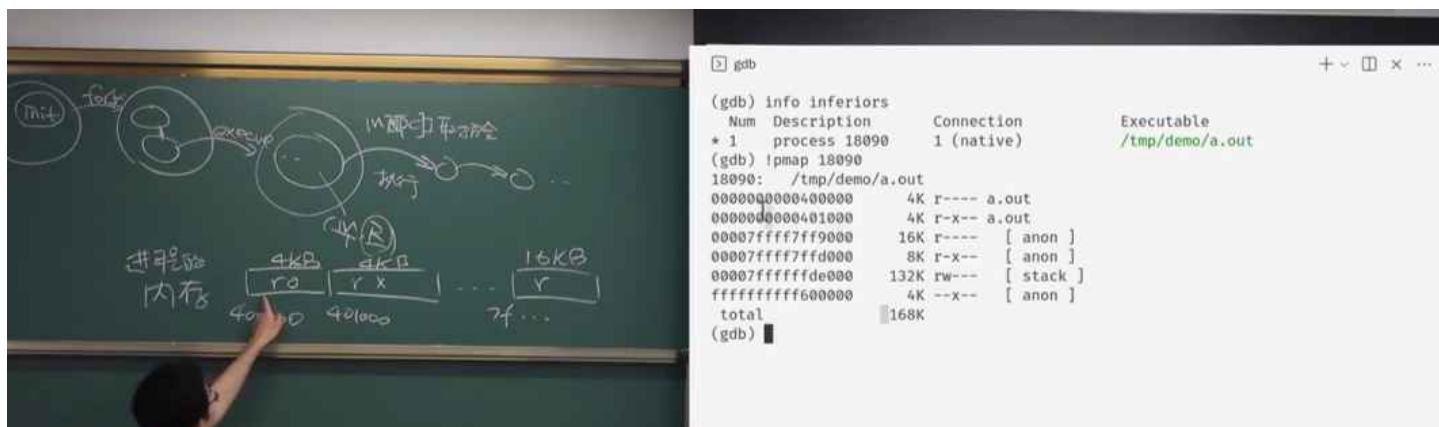
pmap (1) - report memory of a process

- Claim: pmap 是通过访问 procfs (/proc/) 实现的
- 如何验证这一点？

查看进程的地址空间

- 等程序运行起来后 (gdb)，使用 pmap 命令查看地址空间
- 地址空间是若干连续的“内存段”
 - “段”的内存可以根据权限访问
 - 不在段内/违反权限的内存访问触发 SIGSEGV

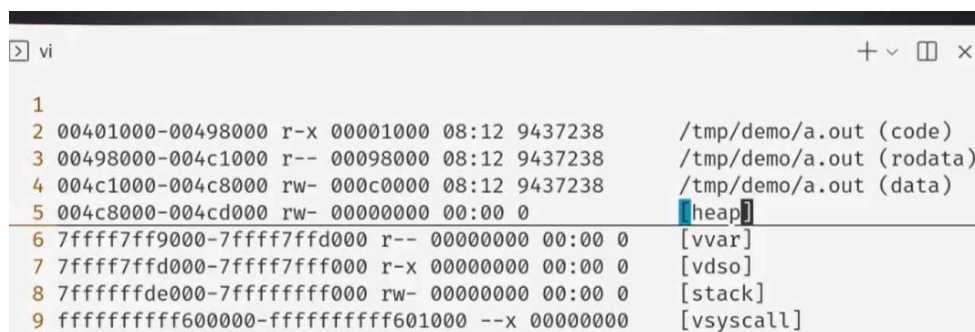
所谓的状态机的状态就是进程的内存，它是由一个一个连续的内存段组成的，这是Linux系统里面最重要的一个抽象。每个区域还带有一个访问权限。



其实系统里面的工具没有什么复杂的，刚才我讲那个小例子，其实有一点这个味道，是我想告诉大家，所有的东西都是通过syscall实现的所谓的操作系统。所谓的操作系统就是进程就一大堆对象。比如说这个 /proc 就是个对象。然后这个 /proc/1234/maps 也是一个对象。

操作系统里面有很多个进程，也是对象，它有进程号，所以你就可以用 kill 去 kill 那个进程号，然后我们的操作系统里面的每一个文件也是对象。然后我们的操作系统为进程提供了API，可以去访问操作系统的

对象，这就是系统调用，所以所有的东西都是用系统调用和操作系统里面的对象来完成的。



1	
2	00401000-00498000 r-x 00001000 08:12 9437238 /tmp/demo/a.out (code)
3	00498000-004c1000 r-- 00098000 08:12 9437238 /tmp/demo/a.out (rodata)
4	004c1000-004c8000 rw- 000c0000 08:12 9437238 /tmp/demo/a.out (data)
5	004c8000-004cd000 rw- 00000000 00:00 0 [heap]
6	7ffff7ff9000-7ffff7ffd000 r-- 00000000 00:00 0 [vvar]
7	7ffff7ffd000-7ffff7fff000 r-x 00000000 00:00 0 [vdso]
8	7ffff7ffde000-7ffff7fff000 rw- 00000000 00:00 0 [stack]
9	fffffffff600000-fffffffff601000 --x 00000000 [vsyscall]

操作系统提供查看进程地址空间的机制

RTFM: /proc/[pid]/maps (man 5 proc)

- 进程地址空间中的每一段
 - 地址 (范围) 和权限 (rwxsp)
 - 对应的文件: offset, dev, inode, pathname
 - TFM 里有更详细的解释
 - 和 readelf (-l) 里的信息互相验证
- 好的本能：做一些代码上的调整，观察 address space 的变化
 - 堆 (bss) 内存的大小
 - 栈上的大数组 v.s. memory error

```
#define MB * 1048576
char mem[64 MB];
```

注意：有些系统调用可以不进入内核，也可以执行；尤其是有些数据，如果是只读的，我只要我的操作系统内核和所有的进程都共享这个数据，对吧？比如说我的系统启动以后，这个系统距离 1970 年 1 月 1 号的秒数。

```
fish
$ ./scripts/askgpt 'What kind of tools can help me print out the address space o
$ ./scripts/askgpt --gpt4 'What are vvar and vdso in my Linux process? Explain in
  detail.'
提问：What are vvar and vdso in my Linux process? Explain in detail.
gpt-4回复：vvar and vdso are special memory regions in a Linux process that are u
sed to improve performance and provide certain kernel-related functionalities to
user-space programs.

1. vvar (Variable Vector):
vvar is a memory region that contains kernel variables that are frequently access
ed by user-space programs. These variables are read-only and can be accessed dire
ctly by the user-space programs without making a system call. This reduces the ov
erhead of switching between user and kernel mode, thus improving performance.

Some of the variables stored in the vvar region include:

- The current time (time_t)
- The page size of the system
- The process's CPU clock frequency

These variables are updated by the kernel, and user-space programs can read them
directly from the vvar region.

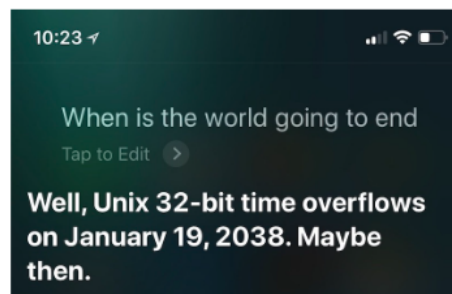
2. vdso (Virtual Dynamic Shared Object):
vdso is a memory region that contains a small shared library provided by the kern
el. This library contains a set of functions that are commonly used by user-space
programs and can be executed directly in user mode, without the need for a syste
```

RTFM (5 proc): 我们发现的宝藏

vdso (7): Virtual system calls: 只读的系统调用也许可以不陷入内核执行。

无需陷入内核的系统调用

- 例子: time (2)
 - 时间：内核维护秒级的时间(所有进程映射同一个页面)
- 例子: gettimeofday (2)
 - RTFSC (非常聪明的实现)
- 更多的例子：问 GPT 吧！
 - 计算机系统里没有魔法！我们理解了进程地址空间的**全部**！



2. 进程地址空间管理

那么你们接下来就一定会想到操作系统应该给我们提供的一个API，可以使我们修改这个进程的内存映射。因为你想我们的操作系统eventually，它是可以支持应用程序分配更多的内存的。我想要 malloc

一个 1GB 的内存是合理合法的，而且我们的操作系统是可以给我们完成的，但是在进程的初始状态的时候，我们的地址空间里面并没有那么多内存。

你看到了初始的时候，我们的二进制文件里面声明了多少内存，我的地址空间里面就有多少内存，不多。所以我们的操作系统就一定会会有一个系统调用或者一个 API 可以帮助我们来修改这个地址空间。

而实现这个syscall就是三个：

- mmap：申请一段
- munmap：删除一段
- mprotect：修改映射权限。比如说我有一段内存，我用 malloc 申请内存，然后我现在想要让这个内存变得可以执行。默认的时候，你申请的内存是不可执行的，因为如果它可执行就有安全隐患，万一你的系统被攻破了，被人入侵了，那他就可以写一段代码在里面就跑过去执行了。所以默认的时候代码是就是数据是不能执行的。

地址空间 = 带访问权限的内存段

操作系统应该提供一个**修改进程地址空间的系统调用**

```
// 映射
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);

// 修改映射权限
int mprotect(void *addr, size_t length, int prot);
```

本质：在状态机状态上增加/删除/修改一段可访问的内存

- mmap: 可以用来申请内存 (MAP_ANONYMOUS)，也可以把文件“搬到”进程地址空间中

关注两个参数：

- fd：文件描述符。所谓的文件描述符，在 Unix 世界里面，everything is a file。所以什么是一个文件描述符？就是一个everything的descriptor。你不要把它理解成是和文件相关，比如说一个文件名或者什么。**文件描述符是 everything 的descriptor，它是一个指向操作系统对象的指针。**
 - 你比如说你要打开一个文件，你会得到一个 file descriptor，然后你要创建一个管道，它会给你一个 file descriptor，然后你要用套接字去连接一台远程的计算机，它也会给你一个 file descriptor。

把文件映射到进程地址空间？

它们的确好像没有什么区别

- 文件 = 字节序列 (操作系统中的对象)
- 内存 = 字节序列
- 操作系统允许映射好像挺合理的.....
 - 带来了很大的方便
 - ELF loader 用 mmap 非常容易实现
 - 解析出要加载哪部分到内存，直接 mmap 就完了
 - 我们的 loader 的确是这么做的 (strace)

使用 mmap

Example 1: 申请大量内存空间

- 瞬间完成内存分配
 - mmap/munmap 为 malloc/free 提供了机制
 - libc 的大 malloc 会直接调用一次 mmap 实现
- 不妨 strace/gdb 看一下

Example 2: Everything is a file

- 映射大文件、只访问其中的一小部分

```
with open('/dev/sda', 'rb') as fp:
    mm = mmap.mmap(fp.fileno(),
                    prot=mmap.PROT_READ, length=128 << 30)
    hexdump.hexdump(mm[:512])
```

Take away messages

状态机的视角自然地将我们引入“内存到底是什么”的问题——它的答案同样也很自然：带有访问权限控制的连续内存段。我们可以通过 mmap、munmap、mprotect 三个系统调用调整状态机的地址空间，包括分配匿名的内存、映射文件内容到内存、修改访问权限等。更有趣的是操作系统有“兼容一切应用程序”的需求，调试器也不在话下——这也给了我们入侵其他进程地址空间的机制。