

# Lab2 system calls

2351759 程琮越

Using gdb (easy)	1
1. 实验目的	1
2. 实验内容	2
3. 问题解决	4
4. 实验心得	4
System call tracing (moderate)	4
1. 实验目的	4
2. 实验内容	5
3. 问题解决	7
4. 实验心得	7
Attack xv6 (moderate)	8
1. 实验目的	8
2. 实验内容	8
3. 问题解决	8
4. 实验心得	8
实验得分	9

## 切换到 syscall 分支

```
cheng@cheng-VMware-Virtual-Platform:~/桌面/xv6-labs-2024$ git fetch
remote: Enumerating objects: 89, done.
remote: Counting objects: 100% (89/89), done.
remote: Compressing objects: 100% (51/51), done.
remote: Total 89 (delta 40), reused 86 (delta 37), pack-reused 0 (from 0)
展开对象中: 100% (89/89), 2.04 MiB | 22.00 KiB/s, 完成.
来自 ssh://ssh.github.com:443/yue-yue-1215/xv6-labs-2024
* [新分支]      main      -> origin/main
cheng@cheng-VMware-Virtual-Platform:~/桌面/xv6-labs-2024$ git checkout syscall
分支 'syscall' 设置为跟踪 'upstream/syscall'。
切换到一个新分支 'syscall'
cheng@cheng-VMware-Virtual-Platform:~/桌面/xv6-labs-2024$ make clean
rm -rf *.tex *.dvi *.idx *.aux *.log *.ind *.ilg *.dSYM *.zip *.pcap \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out user/usys.S user/_* \
kernel/kernel \
mkfs/mkfs fs.img .gdbinit __pycache__ xv6.out* \
ph barrier
```

## Using gdb (easy)

### 1. 实验目的:

本实验围绕 xv6 内核系统调用调试展开，从远程连接 GDB 开始，通过设置断点、单步执行等

操作分析 `syscall` 函数。重点修改代码使程序访问无效地址 0，观察内核崩溃现象，定位故障汇编指令、对应寄存器（如 `a3` 对应 `num`），结合 `scause` (0xd, 加载页故障) 和 `stval` (0x0, 故障地址)，验证地址 0 未映射到内核空间，以此理解内核地址映射、异常处理及系统调用机制，完整掌握内核调试流程与底层原理。

## 2. 实验内容：

1) 输入 `make qemu-gdb`

```
cheng@cheng-VMware-Virtual-Platform:~/桌面/xv6-labs-2024$ make qemu-gdb
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M
ice virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::26000
```

2) 打开另一个终端

先安装多架构 GDB

```
sudo apt update
```

```
sudo apt install gdb-multiarch
```

再调试 `xv6`

```
gdb-multiarch kernel/kernel
```

然后在 GDB 中连接 QEMU

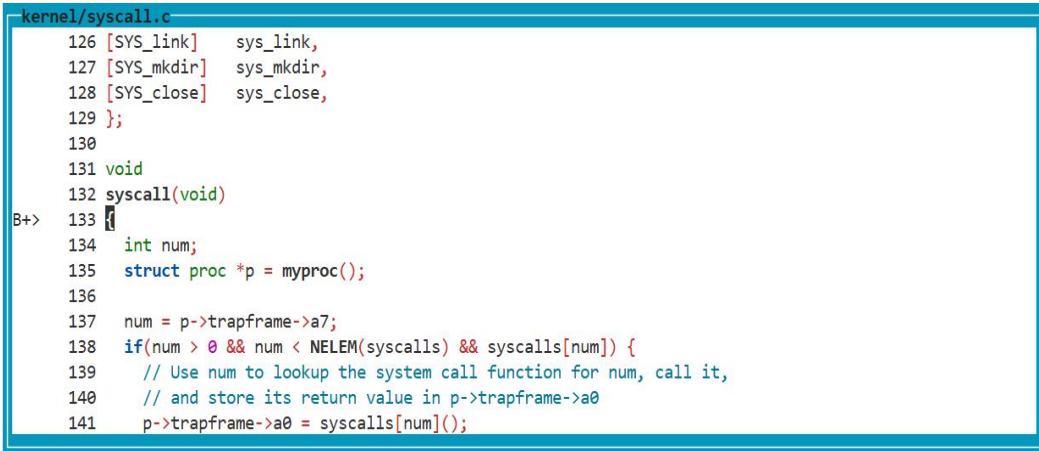
```
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x0000000000001000 in ?? ()
```

3) 设置断点并继续执行直到触发断点

```
(gdb) b syscall
Breakpoint 1 at 0x80001c72: file kernel/syscall.c, line 133.
(gdb) c
Continuing.
[Switching to Thread 1.3]
```

```
Thread 3 hit Breakpoint 1, syscall () at kernel/syscall.c:133
133 {
```

4) 输入 `layout src` 将窗口一分为二，可查看源码布局



```
kernel/syscall.c
126 [SYS_link]    sys_link,
127 [SYS_mkdir]   sys_mkdir,
128 [SYS_close]   sys_close,
129 };
130
131 void
132 syscall(void)
B> 133 {
134     int num;
135     struct proc *p = myproc();
136
137     num = p->trapframe->a7;
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139         // Use num to lookup the system call function for num, call it,
140         // and store its return value in p->trapframe->a0
141         p->trapframe->a0 = syscalls[num]();
```

remote Thread 1.3 (src) In: syscall L133 PC: 0x80001c72

5) 输入 `n` 直到跳过 `struct proc *p = myproc()`; 打印当前进程的结构体

```
(gdb) p /x *p
$4 = {lock = {locked = 0x0, name = 0x800071c8, cpu = 0x0}, state = 0x4, chan = 0x0, killed = 0x0,
xstate = 0x0, pid = 0x1, parent = 0x0, kstack = 0x3fffffd000, sz = 0x1000, pagetable = 0x87f55000,
trapframe = 0x87f56000, context = {ra = 0x800012ae, sp = 0x3fffffd000, s0 = 0x3fffffd000, s1 = 0x8000a670,
s2 = 0x8000a240, s3 = 0x1, s4 = 0x800104f8, s5 = 0x3, s6 = 0x8001b310, s7 = 0x1, s8 = 0x8001b438,
s9 = 0x4, s10 = 0x0, s11 = 0x0}, ofile = {0x0 <repeats 16 times>}, cwd = 0x80018780, name = {0x69, 0x6e,
0x69, 0x74, 0x63, 0x6f, 0x64, 0x65, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}}
```

6) 查看 p->trapframe->a7 的值, 查看 sstatus 寄存器, 查看调用栈

p->trapframe->a7 的值为 0x7, 表示系统调用号, 用于告诉内核用户程序请求执行哪个系统调用

要看 CPU 之前处于什么模式, 可以查看 sstatus 寄存器中 SPP 位 (第 8 位), SPP 位为 0 表示 CPU 之前处于用户模式

```
(gdb) p /x p->trapframe->a7
$5 = 0x7
(gdb) p /x $sstatus
$6 = 0x200000022
```

用 backtrace 可以看到 usertrap () 就是直接调用 syscall 的函数

```
(gdb) backtrace
#0  syscall () at kernel/syscall.c:133
#1  0x0000000080001a2e in usertrap () at kernel/trap.c:67
#2  0x0000000000000000 in ?? ()
Backtrace stopped: frame did not save the PC
```

7) 将 syscall.c 中 num = p->trapframe->a7; 替换为 num = \* (int \*) 0; 会导致内核 panic 要追踪内核 page-fault panic 的来源, 需要搜索刚刚看到的 panic 打印的 sepc 值在文件 kernel/kernel.asm, 其中包含编译的内核

```
cheng@cheng-VMware-Virtual-Platform:~/桌面/xv6-labs-2024$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/
ice virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
```

xv6 kernel is booting

```
hart 2 starting
hart 1 starting
scause=0xd sepc=0x80001c82 stval=0x0
panic: kerneltrap
```

8) 崩溃时执行的汇编指令: lw a3,0(zero), 对应 num 变量的寄存器: a3

```
cheng@cheng-VMware-Virtual-Platform:~/桌面/xv6-labs-2024$ make
make: "kernel/kernel"已是最新。
cheng@cheng-VMware-Virtual-Platform:~/桌面/xv6-labs-2024$ grep -n "80001c82" kernel/kernel.asm
4299: 80001c82: 00002683 lw a3,0(zero) # 0 <_entry-0x80000000>
```

```
(gdb) b *0x80001c82
Breakpoint 1 at 0x80001c82: file kernel/syscall.c, line 137.
```

9) 输入 layout asm, 再输入 c, 程序会停在 lw a3,0(zero) 指令处, 这是崩溃前的最后一条指令。

```
0x80001c78 <syscall+6> sd      s1,8(sp)
0x80001c7a <syscall+8> addi    s0,sp,32
0x80001c7c <syscall+10> jal     0x8000d56 <myproc>
0x80001c80 <syscall+14> mv      s1,a0
B>0x80001c82 <syscall+16> lw      a3,0(zero) # 0x0
0x80001c86 <syscall+20> addiw   a4,a3,-1
0x80001c8a <syscall+24> li      a5,20
0x80001c8c <syscall+26> bltu    a5,a4,0x80001caa <syscall+56>

remote Thread 1.3 (asm) In: syscall
Continuing.
[Switching to Thread 1.3]
```

Thread 3 hit Breakpoint 1, `syscall ()` at `kernel/syscall.c:137`

确认内核 panic 时正在运行的进程的名称及进程 ID

```
(gdb) p p->name
$1 = "initcode\000\000\000\000\000\000\000"
(gdb) p p->pid
$2 = 1
```

### 3. 问题解决:

- 1) 最开始想尝试 `gdb kernel/kernel` (启动 GDB 并加载内核符号) 发现单步执行时出现 (gdb) `c The program is not being run.` 的问题, 安装多架构 GDB 解决问题。
- 2) 修改 `syscall.c` 后未重新编译内核, `kernel.asm` 仍是旧版本, 不包含新的 `sepc` 地址导致 `grep -n "80001c82" kernel/kernel.asm` 在终端执行依然查找不到, 输入 `make` 重新编译可以解决问题。

### 4. 实验心得:

本次实验让我深入理解了 xv6 内核调试流程。从解决 GDB 连接问题到分析崩溃原因, 每步都需细致操作。通过观察寄存器状态和汇编指令, 我直观认识到无效地址访问的后果, 也明白了 `scause`、`stval` 等寄存器在异常处理中的作用, 深刻体会到理论与实践结合对掌握操作系统内核原理的重要性。

## System call tracing (moderate)

### 1. 实验目的:

本实验旨在为 xv6 操作系统添加系统调用跟踪功能。通过创建 `trace` 系统调用, 使进程能设置跟踪掩码, 指定需跟踪的系统调用。修改内核让符合掩码的系统调用在返回时输出进程 ID、调用名及返回值, 并确保掩码可由子进程继承, 最终实现对特定进程及其子进程的系统调用跟踪, 辅助后续调试。

## 2. 实验内容:

1) 添加系统调用编号和用户空间接口

修改 kernel/syscall.h, 添加系统调用编号

```
#define SYS_trace 22
```

修改 user/user.h, 添加函数原型

```
int trace(int);
```

修改 user/usys.pl, 添加系统调用存根

```
entry("trace");
```

2) 修改进程结构以存储跟踪掩码

修改 kernel/proc.h 中的 proc 结构体

```
struct proc {  
    struct spinlock lock;  
  
    int trace_mask; //add
```

3) 实现 trace 系统调用

在 kernel/sysproc.c 中添加 sys\_trace 函数

```
// 新增系统调用: 设置当前进程的跟踪掩码
```

```
uint64 sys_trace(void) {  
    int mask;  
    // 从用户空间获取掩码参数  
    if(argint(0, &mask) < 0)  
        return -1;  
  
    // 设置当前进程的跟踪掩码  
    myproc()->trace_mask = mask;  
    return 0;  
}
```

4) 修改 fork 以继承跟踪掩码

在 kernel/proc.c 的 fork 函数中, 确保子进程继承父进程的跟踪掩码

```
int  
fork(void)  
{  
    int i, pid;  
    struct proc *np;  
    struct proc *p = myproc();  
  
    // Allocate process.  
    if((np = allocproc()) == 0){  
        return -1;  
    }  
  
    np->trace_mask = p->trace_mask; // add
```

5) 修改系统调用处理函数以打印跟踪信息

修改 kernel/syscall.c

首先添加系统调用名称数组并声明函数

```
extern uint64 sys_close(void);  
extern uint64 sys_trace(void); |
```



```
static char *syscall_names[] = {
    [SYS_fork]    "fork",
    [SYS_exit]    "exit",
    [SYS_wait]    "wait",
    [SYS_pipe]    "pipe",
    [SYS_read]    "read",
    [SYS_kill]    "kill",
    [SYS_exec]    "exec",
    [SYS_fstat]   "fstat",
    [SYS_chdir]   "chdir",
    [SYS_dup]     "dup",
    [SYS_getpid]  "getpid",
    [SYS_sbrk]    "sbrk",
    [SYS_sleep]   "sleep",
    [SYS_uptime]  "uptime",
    [SYS_open]    "open",
    [SYS_write]   "write",
    [SYS_mknod]   "mknod",
    [SYS_unlink]  "unlink",
    [SYS_link]    "link",
    [SYS_mkdir]   "mkdir",
    [SYS_close]   "close",
    [SYS_trace]   "trace"
};
```

再修改 syscall 函数

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

    num = p->trapframe->a7;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        // 调用系统调用函数并获取返回值
        p->trapframe->a0 = syscalls[num]();

        // 检查是否需要跟踪此系统调用
        if(p->trace_mask & (1 << num)) {
            // 打印进程ID、系统调用名称和返回值
            printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num], p->tr
        )
    } else {
        printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}
```

然后在 syscalls 数组中添加 sys\_trace 条目

```
[SYS_close]    sys_close,
[SYS_trace]    sys_trace // add
};
```

6) 在 Makefile 的 UPROGS 部分添加 \$U/\_trace \

7) 进行验证

```

$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 971
3: syscall read -> 298
3: syscall read -> 0
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 971
4: syscall read -> 298
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$

$ trace 2 usertests forkforkfork
usertests starting
7: syscall fork -> 8
test forkforkfork: 7: syscall fork -> 9
9: syscall fork -> 10
10: syscall fork -> 11
11: syscall fork -> 12
10: syscall fork -> 13
11: syscall fork -> 14
10: syscall fork -> 15
11: syscall fork -> 16

cheng@cheng-VMware-Virtual-Platform:~/桌面/xv6-labs-2024$ ./grade-lab-syscall trace
make: "kernel/kernel"已是最新。
== Test trace 32 grep == trace 32 grep: OK (1.2s)
== Test trace close grep == trace close grep: OK (0.9s)
== Test trace exec + open grep == trace exec + open grep: OK (1.3s)
== Test trace all grep == trace all grep: OK (0.8s)
== Test trace nothing == trace nothing: OK (1.1s)
== Test trace children == trace children: OK (19.6s)

```

### 3. 问题解决:

- 1) 在 kernel/syscall.c 顶部位添加添加函数声明, 导致未定义的错误。
- 2) 格式字符串与参数类型不匹配的问题, %d 去格式化一个 uint64 类型的变量导致输出结果不正确, 修改前为 printf("%d: syscall %s -> %d\n", p->pid, syscall\_names[num], p->trapframe->a0); 修改后为 printf("%d: syscall %s -> %ld\n", p->pid, syscall\_names[num], p->trapframe->a0)。

### 4. 实验心得:

完成 xv6 系统调用跟踪实验, 让我深入理解了系统调用的实现流程。从添加系统调用编号、用户接口, 到修改进程结构存储掩码、实现跟踪逻辑, 每步都需兼顾内核与用户态交互。fork 时继承掩码的设计, 让我体会到进程间状态传递的细节, 也明白了调试工具背后的内核机制, 提升了对操作系统运行的理解。

## Attack xv6 (moderate)

### 1. 实验目的：

本实验旨在通过修改 `user/attack.c`，利用 xv6 内核的内存分配漏洞（新分配内存保留前使用者数据），提取 `secret` 进程退出后遗留的 8 字节秘密并输出到文件描述符 2。以此理解内核漏洞如何破坏进程隔离，体会系统安全中内存清零的重要性，掌握漏洞利用的基本思路。

### 2. 实验内容：

1) 漏洞环境已经预先搭建完成，`#ifndef LAB_SYSCALL` 使清零代码被自动排除，从而实现“内存分配后保留之前的内容”的漏洞效果。

2) 先使用 `sbrk(PGSIZE*17)` 分配了 17 页内存（实际使用的内存应该是与第一段代码中分配的内存区域重叠）；然后通过 `end + 32` 来定位到第一段代码写入 `argv[1]`（即传入的秘密密码）的内存地址；最后将 8 字节秘密写入文件描述符 2（`stderr`）。

```
$ attacktest
OK: secret is cc/cb..
```

3) 要使攻击有效，必须确保密码数据在 `end + 32` 这个位置，因此不能将 32 改为 0。如果修改为 0，攻击就会失效，因为代码的内存布局和攻击的预期位置都发生了变化。

### 3. 问题解决：

1) 出现 `FAIL: no/incorrect secret` 错误，若未准确找到 `secret` 进程存储秘密的位置，会导致读取到无效数据。需确认 `secret` 进程的内存布局，确保偏移量正确。

2) 如果传入的 `argv[1]`（即密码）过长，可能会导致溢出，并覆盖临近的内存区域，这会破坏其他数据。如果常量字符串的结束位置紧挨着密码存储的位置（`end + 32`），那么密码的存储可能会覆盖常量字符串，导致数据丢失或错误。

### 4. 实验心得：

本次实验让我深刻体会到内核细节对系统安全的重要性。内存未清零的微小漏洞能让进程突破隔离获取敏感信息，说明了内核开发需注重细节，任何看似无关的操作（如内存清零）都可能成为安全隐患。同时，我也理解了漏洞利用的基本逻辑，为今后学习系统安全打下基础。



## 实验得分

```
== Test answers-syscall.txt ==
answers-syscall.txt: OK
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (3.6s)
== Test trace close grep ==
$ make qemu-gdb
trace close grep: OK (0.3s)
== Test trace exec + open grep ==
$ make qemu-gdb
trace exec + open grep: OK (1.2s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.9s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (19.2s)
== Test attack ==
$ make qemu-gdb
attack: OK (0.3s)
== Test time ==
time: OK
Score: 50/50
```