Assignment 6

CS 6960, Fall 2017

Due: October 5, 2017

Alan Humphrey

Do one of the following

1. Make ˆC (control-C) do something sensible, such as kill the current foreground process (and perhaps all of its children?).
2. Pass a bad pointer and/or an over-long length argument to the read() system call, causing the OS to die with the "unexpected trap" message. If you can do this you are awesome and get a super bonus prize of some sort. If you can't do this, explain how xv6 prevents it.
3. Explain how the arrow keys change the cursor position.

Choice: Item 2 - try and crash the kernel via `read` system call.

1 Claim

Based on the problem spec, my approach looks directly at the `sys_read()` call in `sysfile.c` (Listing below), and attempts to find any way to get a bad argument into the kernel via this sys call. Through initial efforts (certainly not exhaustive), including "over-long" length arguments, junk pointers, etc I never could trigger the execution of:

`cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)")` in `trap.c`

My argument then becomes one of explaining how xv6 prevents this from happening, essentially showing how the error checking within `sys_read()` guards sufficiently against bad arguments from user space.

2 Argument

Looking at `sysfile.c`, specifically at `sys_read()`, we see three fundamental error checking functions (listing from syscall.c below.

```
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
```

Listing 1: sysfile.c and listing of sys_read(void)

For any call to `sys_read()`, the arguments are checked as:

- `argfd(int n, int *pfd, struct file **pf)` - seems to adequately protect against bad system call numbers and associated file descriptors

- `argint(int n, int *ip)` - in turn calls `fetchint(uint addr, int *ip)` to get the system call, which checks the specified addr against the size of process memory (bytes).

- Finally, the third segment of the error checking calls `argptr(int n, char **pp, int size)`, where a check is performed to see that the provided pointer lies within the process address space.

A complete listing of these error checking functions is provided on the following page.

3 Conclusion

Though I was NOT able to trigger the unknown trap message, it would not surprise me if someone is able to figure out some way to do this, as my approach was certainly not exhaustive/extensive. All-in-all, it looks like xv6 does an OK job at guarding against writing to pages outside of a particular process' address space.

```c
// Fetch the nth word-sized system call argument as a file descriptor
// and return both the descriptor and the corresponding struct file.
static int
argfd(int n, int *pfd, struct file **pf)
{
  int fd;
  struct file *f;

  if(argint(n, &fd) < 0)
    return -1;
  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;
  if(pfd)
    *pfd = fd;
  if(pf)
    *pf = f;
  return 0;
}

// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}

// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size bytes.  Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  int i;
  struct proc *curproc = myproc();

  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

Listing 2: syscall.c and listing of error checking functions