CS61 Scribe Notes
Lecture 18 - 11/6/14
Fork, Advanced Virtual Memory

Roger, Ali, and Tochi

Topics:
exploits
fork
shell programming

rest of course announcements/ending (for later info)

- final (not as time pressed as midterm hopefully)
- reading period:
    - turn in psets again if unhappy with prior performance for updates. Can possibly bump up grade for whole course. Programming ability is measured by psets, so it's important
    - pset6: due last day of class but can turn in later…?
    - NEW: anyone can sign up for 15 min w/ eddie or tf to talk thru code style feedback

os02
make run
4 options: alloc recurse fork panic
recurse (added functionality last time. Auto-growing stack)
p-alloc: page allocator process (like pset)

p-alloc.c
rem symbol defined to be last address
set as default for unix based prog
/* digression on linkers

- look at link directory of os's, has script that tells linker how to smash object files together into executable. Can build single binary out of bunch of obj files.
- proc-start-adder. Told linker. (see . = 0x100000 part of code)
- then tell others ab obj files, smash all together into .text segment
- etext defined at that point
- look @ readonly data, put that together

- then have page of read only data, and read write data on diff, page aligned offset
- PROVIDE(end = .); //tells linker to put end
- very simple linker script. If want diff pics where shared data after readwrite data, could just change order of linker to change how binary is built.
- get underneath abstraction barriers.
- 1 lvl below gcc

*/
start on first heap page
call sys_page_alloc(data);
allocates page of mem at that address
then write on that page
if succeed, say allocated memory is writable
check if writes work by comparing the data wrote
if see problem, panic.

key part of implement is in kernel
complete implement in alloc free page…
case INT_SYS_PAGE_ALLOC
page_alloc_unused
permissions gived are PTE_P |PTE_W|PTE_U
if succeed, return 0
if rerror, return -1

if get rid of PTE_W and map mem, program will pagefault. Write protection problem.

kernel can crash w/…
(latent bug) p-alloc could address really low address and mess w/ kernel mem
invariants need to be maintained for kernel to be isolated
    kernel mem has to be mapped correctly
    kernel mem is how sys call happens (switch to priv, but don't change page symbol)
when kernel runs in response to exception, address it runs at is virtual. every address in sys is virtual. so interrupt descriptor table uses physical memory, but ones inside of it are virtual addresses
address of kernel stack is a virtual address
so if screw up virtual address containing exception handlers, auto-fail

change sys_page_alloc(data) to sys_page_alloc((char*) 0x40000) //changes to kernel itself addr, will result in panic within kernel.
k-hardware.c assertion: l1 pte pte fail
b panic
bt shows virtual_memory_map is getting lost. something is referencing adddress 0x00006000. Crazy crap happens because we gave a crazy address to sys_page_alloc().

can fix w/ INT_SYS_PAGE_ALLOC
can make sure deputy stays non-confused by adding in if statement that wantva >= PROC_START_ADDR. Page is physical address of page (only NULL of run out of physical mem). In tis case, wantva (virtual) matters. If portion of address space allowed.
	now, will have user-level panic. Process/attack fails, not kernel.
Process's page table is in force when exception occurs. Could change inside kernel, but at first run w/ processor page.

(if pass in NULL to syscall last check), will return -1

if pass in huge #, will fail virtual memory map
if pass in a page address that's not page aligned (data+1)
	HARDWARE PANIC
whenever user passes data, must validate it carefully.
Process isolation saves you from having to consider everything as malicious
kernel callers are processes, which are isolated. so kernel has to verify on every syscall that inputs are not crazy. if kernel does something wrong w/ crazy inputs, fail

go fix bug @ INT_SYS_PAGE_ALLOC
add if condition for wantva % PAGESIZE == 0 (//means wantva must be page aligned)
//should also probably check if address is small enough
// now fully isolated
other invariant: addresses must be page aligned

side note: to avoid a memory leak if user gives bad argument, you can check the user's argument before doing anything. Avoid doing something with bad data.

CVE: Common Vulnerabilities and Exposures Database

- List of many serious bugs, reported to government
- search for particular vulnerability: cve-2014-0038
    - 1 of 14 privilege escalation vulnerabilities reported for linux in past year
    - can use a privilege vulnerability to gain unauthorized privilege to root
- links to dark corner of internet: place where people write bugs for fun
    - particular bug:
    - convinces kernel to remap memory so that user can access other data
    - triggers vulnerability by accessing code user shoved in
    - gives that code privilege to run the system

No more attacks, more productive

Fork (part 5 of pset) (and a bit for next pset)

- OS purpose is to run more than 1 process
- How to:
    - start a process:
    - exit a process
    - communicate
- pid_t fork() (Fork is one of the few times we have a function that returns twice)
    - copies current process
    - returns: 0 to copy, <ID of new process> to original
    - About 20 lines to implement
    - One of the most confusing system calls there is
        - seems huge
        - takes one process and turns it into 2
        - copies all of processes resources
        - the fact that it works on large system is magical :)
- Hypothetical thinking:
    - new_process system call
    - returns process ID of a new empty process
    - what are the consequences?
        - need some mechanism for the old process to put stuff into the old process's memory and initial registers (stuff that kernel fundamentally owns)
        - seems like this is good since we can set up the new process exactly how we want it (since doesn't depend on anything else)
        - this seems dangerous since processes are supposed to be isolated, not have access to other processes' memory

        - this introduces a new idea: un-isolated process (process whose memory is empty and relies on other process to initialize it)
    - Memory is allocated by getting blank memory, not copying. Alloc a new process by copying
    - Other idea: create basic default process that runs already
        - clone (fork) knows what to do because it is in a particular context (parent process's memory state)
- Evolution of Unix Timesharing paper (see course website)
    - system built on machine with 32 KB or less (so tiny!)
    - could run exactly 2 processes: shell and 1 slot
    - failed when tried to make shell make other process
    - fork required 27 lines of assembly code (similar to our implementation)

Fork - What are the resources?

- WeensyOS, what do we need to copy?
    - copy process memory (stack, heap, data, text)
    - copy parent process pagetables
    - copy registers
- What don't we copy (dif between parent and child)
    - there are some things within a process that are dif that process doesn't know
    - Physical addresses: parent and run on different physical memory pages
    - process can't tell what physical page it's on (only knows virtual address)
    - definitely different:
        - process ID (sys get pid)
        - eax register (one has eax=0, other has eax>0)

```
pid_t p = fork ();
if  (p==0) {
        /*child*/
}
else { /*parent*/
        if (p<0) {
                /*error*/
        }
        if (p>0) {
                /*have child*/
        }
}
```

Let's leave weensyOS for a bit.
Go into lecture 18 directory. fork1.c

- both processes are printed to stdio.
- first of all process ids look like they are allocated sequentially.
- second thing noticed is that the message from the first process is always printed before we get a new prompt.
- Did we get a new terminal? NO
- This is another deep aspect of process isolation. Its goal is to give a process the illusion that it is in complete control of the machine.
- Machine has primary memory and other devices. Processes on the machine share the same IO devices. They are the way processes can communicate. They don't communicate over memory or CPU time.
- Memory for devices are files. Examples of files are stdin stdout and stderr
- When we run fork1 we start out with a process.
- 0 is stdin 1 stdout and 2 is stderr. The shells file descriptor table contains these values.
- In this program there is one device that matters which is the console. All of the std(in out err) are linked to the console.
- we type ./fork1 at the shell: what happens?
    - How to figure out what happens: strace!
    - strace -o strace.out sh //runs a shell inside of strace
        - run strace on fork1 and look at output
        - the shell waits for fork1 to complete, then gains control after.
        - creates a new process that is a copy of the shell, and has copy of shell's file descriptor table
        - looking at strace we don't see the child.
        - there is a flag -s that lets us track all children.
        - Every line in this kind of strace is prefixed by the process id that implemented that system call.
        - When we execute a command at the shell we expect the shell to let that run.
        - A child in Uniix cannot wait for its parent to die, but a parent can.
        - execve()
        - exec system calls replace current process with another process loaded in from disk.
    - first thing is done that ./fork1 is read by the shell.
    - exec: replaces process memory with a fresh program image from disk.
    - why do we nee exec? we need exec so that we can run more than one program.

    - However, it should be possible for a process to load a program from disk by itself.
    - The shell is waiting for the child; the child has been replaced by fork 1; the child has created a new child that's running fork 1 with a copy of the pagetable.
    - The child dies; shell begins and prints a prompt; at that point, second child boots up (prints to same screen b/c file descriptors point to same IO), runs, and then prints output after already existing shell prompt
    - Why was second child created? The second child was created because fork1 called fork.
- Output results of fork1 to an output file f.
    - shell read a command from stdin
    - shell then calls open that creates the output file
    - when we open a file, that adds a descriptor to our file descriptor table.
    - we start a child and the child actually has all previous files.
- a file descriptor table creates a universe in which all processes run. The code that sets up the universe is the code that executes between the fork and exec. This is were the magic of unix really happens and we will get into that next time.
- Once fork1 starts running it has no idea of what was happening in memory.