CS140
Operating Systems
Final – December 12, 2007

OPEN BOOK, OPEN NOTES

Your name: ______________________________

SUNet ID: ______________________________

In accordance with both the letter and the spirit of the Stanford Honor Code, I did not cheat on this exam. Furthermore, I did not and will not assist anyone else in cheating on this exam.

Signature: ______________________________

The exam has 15 questions totaling 100 points. You have 180 minutes to complete them. Some questions may be much harder than others.

I Virtual machines

1. [5 points]:

Consider an x86 virtual machine monitor (VMM) such as VMware that uses binary translation for guest OS kernel code. Some instructions can be translated *identically*, meaning the exact same instruction is executed regardless of whether the OS is running on the VMM or on raw hardware. Other instructions must be translated non-identically. (For example, since the VMM must intercept all hardware interrupts, the `lidt` instruction which loads the address of interrupt descriptor table cannot be translated identically.)

Which of the following instructions must be translated non-identically?

Circle all that apply. There may be more than one answer.

A. `ret`

B. `movl _global, %eax`

C. `jmp *%eax`

D. `call printf`

E. `push %cs`

Answer: A, C, D, and E

2. [10 points]:

Hardware MMUs translate virtual to physical page numbers in accessed memory addresses. A virtual machine monitor must add another layer of translation, mapping VM "physical" addresses (which are now virtualized) to real machine addresses.

One common optimization is to point the hardware MMU at *shadow page tables* that directly map virtual to machine addresses. The VMM computes shadow page tables based on guest OS page tables and its own "physical"→machine page mappings.

The VMM can intercept updates to guest OS page tables by making them inaccessible in the shadow page table. Is it ever safe *not* to protect guest OS page tables and just to let the guest OS directly manipulate its page tables in memory?

Circle one then justify your answer:

Yes, it can be safe **No, it is never safe**

Answer: Yes, as long as none of the virtual addresses covered by the page table are currently accessible in the shadow page table. The guest OS might make some previously inaccessible page valid. Accessing the page would then cause a hidden page fault to the VMM, at which point the VMM can fix up the shadow page table and transparently resume execution.

3. [6 points]:

VMware ESX saves memory by identifying multiple copies of the same physical page and mapping them all copy-on-write to a single machine page. This technique saves memory when, for example, two virtual machines are running the same guest operating system, as the pages containing kernel code will be identical across the two VMs.

Could this technique ever save memory when a single virtual machine is running (assuming the VM is configured to see more "physical" memory than there is machine memory)? Under what circumstances might a modern operating system have multiple physical pages with the same contents?

Answer: Yes. For example, if you copy a file, both the old and new file blocks may be stored in the buffer cache. VMware ESX could consolidate these identical pages.

4. [6 points]:

Give an example of a kernel instruction that can run faster under VMware than on real hardware. Explain why.

Answer: One example is the `cli` instruction, which is slow on real hardware but just requires updating a flag in VMware.

Another example might be I/O to virtual rather than real hardware. An `inb` or `outb` instruction that goes across an I/O bus to a real network card is very slow. But under VMware you are interacting with a virtual network card, and might just be talking to another VM on the same physical machine which would require no real I/O. Thus the `inb` or `outb` gets translated to something much faster.

II Security

5. [6 points]:

The Unix setuid facility allows programs to run with greater privilege than the user who invoked them. The kernel assigns each process both an *effective* and a *real* user ID. Ordinarily these two user IDs are the same. However, when a process executes a setuid binary, it takes on the effective user ID of the binary file's owner, while retaining the real user ID of the invoking user.

Most access control checks performed by the kernel use only the effective user ID. However, process A can kill process B if A's effective user ID matches B's real user ID, even if B's effective user ID is something different. This ensures users can kill any setuid programs they run.

The same logic does not apply to `ptrace`, the debugging system call. Explain what would go wrong if A could attach to B using `ptrace`.

Answer: A could completely control B's memory, registers, and program counter. It could cause B to execute another program, thereby letting A do essentially anything it wants with B's effective user ID. Thus, a user could do essentially anything it wants under the effective user ID of any setuid executable it can invoke.

6. [6 points]:

A Unix operating system ships configured to run this command every night:

```
find /tmp -atime +3 -exec safeunlink {} ;
```

The effect is to find every file that has not been accessed in more than three days under the directory `/tmp` and run the `safeunlink` command on that file's pathname. `safeunlink` deletes a file. Its implementation is below.

Assume `path_to_components` takes a Unix pathname separated by "/" characters and splits it into an NULL-terminated array of strings, one for each component directory.

```
 1  int
 2  main (int argc, char **argv)
 3  {
 4    int i;
 5
 6    char **components = path_to_components (argv[1]);
 7    if (components[0] == NULL)  /* bad path name */
 8      exit (1);
 9
10    for (i = 0; components[i+1] != NULL; i++) {
11      struct stat s1, s2;
12
13      /* Check the type of file components[i].  If it is anything other
14       * than a directory (including a symbolic link), return failure */
15      if (lstat (components[i], &s1) < 0 || !S_ISDIR (s1.st_mode))
16        exit (1);
17
18      if (chdir (components[i]) < 0)
19        exit (1);
20
21      /* If, after changing directory, "." is not the same directory
22       * as components[i] in the parent, then return failure */
23      if (stat (".", &s2) < 0
24          || s1.st_dev != s2.st_dev || s1.st_ino != s2.st_ino)
25        exit (1);
26    }
27
28    return -unlink (components[i]);
29  }
```

What is the purpose of lines 23–25?

Answer: To ensure no symbolic links are traversed while unlinking the file. This avoids the TOCTTOU bug we saw in class where an attacker deletes `/etc/passwd`

7. [6 points]:

Designers of Mandatory Access Control systems strive to reduce *covert channels*, which are mechanisms that can be exploited to transfer information in violation of the system's security policy.

Imagine a system with a Bell-LaPadula-like security policy and two classifications, *secret* (used for sensitive national security information) and *public* (used for publicly releasable information). The policy allows users to read public files while preparing secret documents, but not vice versa.

Suppose that a design limitation in the OS's file locking support creates a well-known high-bandwidth covert channel. Describe a scenario in which a popular but malicious software vendor can exploit this channel to steal secret data.

Answer: The attacker might release *Trojan horse* software that appears to do something useful, but actually leaks information. For example, the software might appear to be a useful enhancement to the emacs text editor. However, when run by a user with a secret security clearance, this software will fork off a *high* process that sets its current-level to secret and a low process that keeps its current-level at public. The high process will then communicate the contents of secret files to the low process, which then in turn leaks them over the network.

8. [5 points]:

The Unix login program checks whether or not a user has entered the correct password before taking on the user's ID and executing the user's shell. Explain how it can check passwords this way without storing the user's actual password on the system?

Answer: It stores $\{\text{salt}, H(\text{password}, \text{salt})\}$, where H is a function that cannot be inverted in a computationally efficient way.

9. [5 points]:

Suppose an attacker breaks into a Unix machine, obtains root (superuser) privileges, and manages to keep them for a long period of time (e.g., many months). What might the attacker do to learn users' real passwords, even if they aren't stored on the system?

Answer: Replace `login` with a malicious version that records the passwords users enter in a file where the attacker can read them.

III Memory management

10. [5 points]:

Explain why a reference-counting garbage collector may leak memory that a stop-and-copy garbage collector would properly collect.

Answer: A data structure may be unreachable from any registers or global pointers, yet have a cycle. E.g., A has a pointer to B, B has a pointer to C, and C has a pointer to A. In such a situation, A, B, and C will all have reference count of 1 and remain uncollected, while the stop-and-copy collector would not reach these objects and thus not copy them to the new heap.

11. [5 points]:

Describe program behavior under which a stop-and-copy garbage collector might give better performance than a reference-counting garbage collector.

Answer: A program that does not allocate or free much memory, but makes lots of copies of pointers will perform better under stop-and-copy. Copying a pointer under stop-and-copy has the same cost as copying any other value (such as an integer), while with a reference-counting scheme one must constantly increment and decrement reference counts.

12. [10 points]:

Define the *total dynamically allocated memory* as the sum of sizes of all `malloc`ed memory regions that have not been `free`d.

Suppose the page size on your machine is 4 KB. Describe a program that never exceeds 4,096 KB of total dynamically allocated memory, yet by calling `malloc` and `free` still requires at least 5,120 KB of virtual memory to run (in addition to memory for code and statically allocated data), regardless of the memory allocator in use.

Hint: Your program can depend on the value of the pointers returned by `malloc` to act in the worst possible way regardless of the `malloc` implementation.

Answer: First, allocate 2,000 2 KB chunks. These will span some number of virtual pages—at least 1,000.

For all such virtual pages, consider the first byte contained by one of the 2 KB allocated chunks. Free all chunks that do not contain such a first byte.

Say you have n chunks that you have not freed. $n \geq 1,000$, since at most half the chunks could have been freed. Moreover, there are at least n pages of memory in use, since each chunk used the first allocated byte on some page.

Now allocate $(2,000 - n)/2$ 4 KB-sized chunks.

IV File systems

13. [5 points]:

Using Berkeley FFS, you have a file system that contains one very large (4 GB) file, and a directory tree with 400,000 files of 10 KB each. You notice that you can stream through the large file, reading the whole thing sequentially, in about 2 minutes. However, a program that traverses the file system and reads all the small files takes almost an hour to run, even though that also involves reading about 4 GB of data. Explain the difference in performance between the two workloads.

Answer: Reading each file requires a disk seek and on average half a rotation, which together might take around 8 msec, while reading the large file sequentially goes at disk bandwidth which might be around 40 MB/sec.

14. [10 points]:

Suppose you modified the Berkeley fast file system (FFS) to remove all synchronous or immediate writes to disk. In the modified file system, all writes to disk are delayed 30 seconds (or until the kernel needs to reclaim the buffer for another purpose). Thus, two successive writes to the same inode, directory block, or other metadata structure usually only require a single write to disk. Moreover, the kernel can order the writes (or ask the disk to order them) in such a way as to get good disk arm scheduling.

Describe a way in which, after a power failure, a directory block may contain, instead of directory entries, arbitrary data from a user's file. Be sure to specify the exact series of system calls made before the power failure in your scenario.

Answer:

unlink – delete file f

mkdir – create new directory d; the kernel re-uses a block that used to belong to f to store directory data

crash – if f's inode has not been cleared yet, but d's inode has been written to disk, f and d will share a cross-allocated block.

15. [10 points]:

Consider the following program:

```
#define FILE "tmp"

int
main (int argc, char **argv)
{
  int i, fd;
  for (i = 0; i < 100; i++) {
    fd = open (FILE, O_CREAT|O_WRONLY|O_TRUNC, 0666); /* create FILE */
    close (fd);
    unlink (FILE);                                    /* delete FILE */
  }

  return 0;
}
```

Is the total number of disk writes required by this program (including writes delayed until after the program exits) likely to be more on XFS (the journaling file system we discussed in class) or FFS with soft updates?

Circle one then justify your answer:

XFS requires more writes **soft updates requires more writes**

Answer: XFS requires more writes, because each create and delete must be entered in the log. Soft updates doesn't need to do any disk writes if you quickly create and delete a file.