CS360 Exam #1: October 19, 1994

Answers

Jim Plank

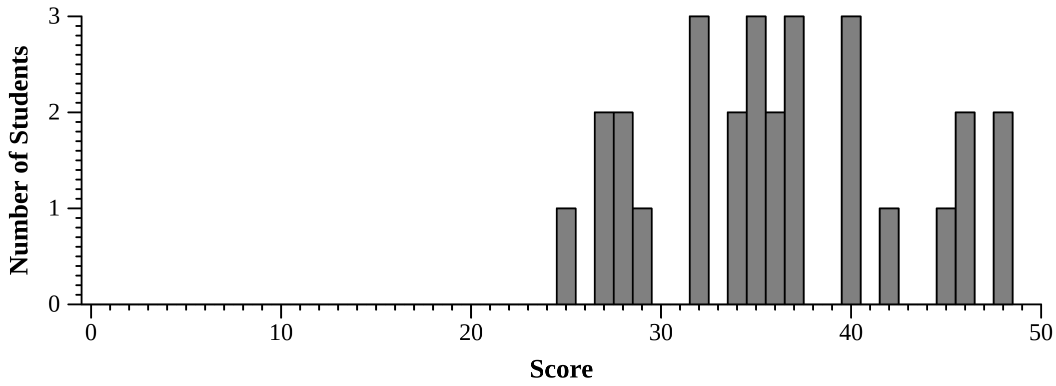
Grading and general comments

So, this test really had two parts — two easy questions (1 & 4), and two more difficult questions (2 & 3). Basically, what I had in mind when I made the test was this: If you have done enough studying and learned enough basic material, you should get at least a 28 on this test. That's 22 points for questions 1 & 4, three points for regurgitating the definition of the setuid but, and 3 points for finding the bug in the the program in question 3. Fortunately, all of you are pretty much at that level. Another 12 points (Parts 2 of questions 2 & 3) made you do some thinking: In question 2, you need to apply the definition of the setuid bit. In question 3, you need to be able to show you understand enough about assembler to write down something reasonable. The remaining 10 points required you think like a systems programmer, and show some creativity. I didn't expect the majority of the class to answer those well, although I did expect to see some clever and unusual answer, and I wasn't disappointed.

Anyway, the bottom line is that no one is failing this class — you all showed that you have learned a good deal of basic material, and that's nice. If I were to assign grade levels to scores – they would be as follows:

- 40 - 50: A
- 30 - 39: B
- 25 - 30: C

These letter grades shouldn't make anyone panic — I would say that no one in this class will be unable to get a final grade of A because of this test. On the whole, I was very pleased with the results.



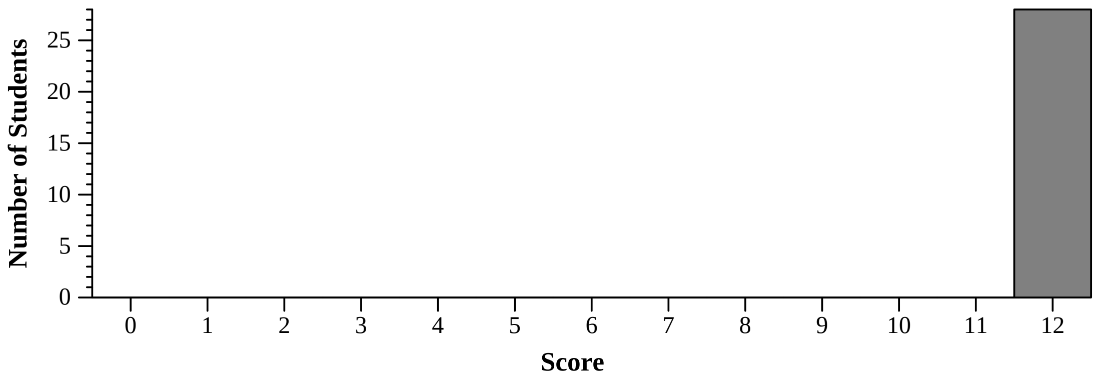
Total Score Histogram

Question 1: 12 points

Buffering file I/O means allocating a large array (called a buffer), and reading from/writing to that array instead of directly from/to the operating system. For reading, when the buffer is empty, it is read from the operating system. For writing, when the buffer is full (or we're done with the file), it is written to the operating system. Buffering is a good idea, because the alternative yields bad performance. If you want to read/write small chunks, then without buffering, you must make a system call for each chunk. Since system calls are expensive, this is a bad strategy.

The standard I/O library in C (for example, `fread()`, `fwrite()`, `getchar()`, etc), implments buffering of file I/O.

Grading: Everyone received full credit for this question. Good job y'all.



Question 1 Histogram

Question 2: 14 points

Part 1 (3 points): The setuid bit is part of a file's protection bits. When set on an executable file, then whenever any user executes this file, he/she does so with the user ID of the owner of the file. (Actually, this only works if the user is logged into the machine whose disk physically owns the file). The setuid bit is used to grant other users restricted access to his files beyond what the normal protection bits give you. For example, if I want other users to be able to write to a log file, but I don't want that file to be writable by the world, I can write a program to update the log file, and set the setuid bit.

Part 2 (6 points): When I do:

```
UNIX> cp /bin/rm .    ; chmod 04755 rm          ; sleep 3600 ; /bin/rm rm
```

Then whenever you run `/mahogany/homes/plank/rm`, you are running `rm` as me. This means that you can delete any of my files that are protected as writable by me. If you run `/mahogany/homes/plank/rm -rf`, you can do even more damage. The bottom line is that you can use this `rm` to delete my files.

```
UNIX> cp /bin/sh .    ; chmod 04755 sh          ; sleep 3600 ; rm sh
```

During the hour that `/mahogany/homes/plank/sh` has its setuid bit set, you can run a shell as me. This means that you can do anything you want as me – remove files, overwrite them, post news, send evil mail to Dr. Poore to get me fired, whatever.

```
UNIX> cp /bin/chmod . ; /bin/chmod 04755 chmod  ; sleep 3600 ; rm chmod
```

During the hour that `/mahogany/homes/plank/chmod` has its setuid bit set, you can run `chmod` as me. This means that you can run `chmod 0777` to any of my files, and delete them with `rm`, or overwrite them with vi.

```
UNIX> cp /usr/local/bin/gcc . ; chmod 04755 gcc ; sleep 3600 ; rm gcc
```

During the hour that `/mahogany/homes/plank/gcc` has its setuid bit set, you can do damage, but not as easily as wehn `sh` was set. Basically, you can use the `-o` option to make `gcc` overwrite my files. If you wanted to be sneaky, you could compile malignant C programs and put them into my `bin` directory with names of common programs (like `cat` and `ls`), so that when I log in and try to run the standard `cat` and `ls`, I'll run the bad versions. This is a good example of a "trojan horse" – I especially liked this answer, as I wasn't expecting it. The answer that was wrong was "I can use `gcc` to compile my own programs as you and then run them as you." That's not correct, since the resulting program from the compilation will *not* have the setuid bit set, so you'll just be running it like a regular program of mine.

Part 3 (3 points): By "forever", I meant forever. And ever and ever. Many of you said "I can use `sh` to change the protections of files, and that will last after the hour is up." Indeed true, but that's not forever – it's just until I notice the problem and `chmod` the programs back. Better answers were one of the following (these got full credit):

- "I mess up your `.cshrc` file so that you can never log in again." Not bad, as I'll probably need sysadmin help to log in again. However, it's not permanent, as I can discover it before logging out, and fix it up, and after I get the sysadmin's help, you can't be malignant any longer.
- "I run your shell and never exit – Therefore I can keep messing you up for as long as I run." Clever answer, which came from a bunch of people. Still, if I turn the machine off, the shell goes away, and you can't be malignant any longer.
- "I edit your `.rhosts` file so that I can always log into your account." Devious, but curable.

- (and this is the real correct answer): Suppose my username is **evilone**. I do the following:

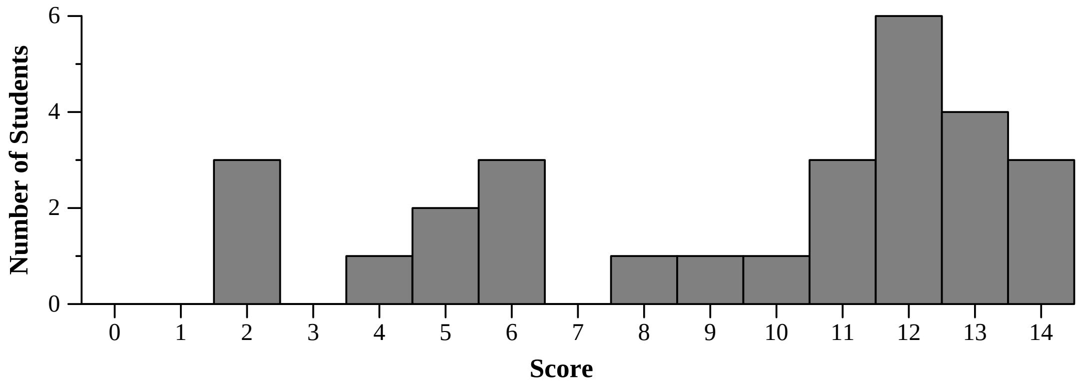
```
UNIX> cd ~/mydirectory
UNIX> pwd
/mahogany/homes/evilone/mydirectory
UNIX> chmod 0777 .
UNIX> ~plank/sh
$ cp /bin/sh /mahogany/homes/evilone/mydirectory
$ chmod 04755 /mahogany/homes/evilone/mydirectory/sh
$ exit
UNIX> ls -l
-rwsr-xr-x  1 plank     106496 Oct 20 14:10 sh
UNIX> chmod 0700 .
```

  What does this do? It basically copies **sh** to **evilone**'s directory, but **sh** is owned by **plank** and has the setuid bit set. Which means that whenever **evilone** wants, he can fire up a **sh** as me. This is truly permanent.

Part 4 (2 points): Most of you said the same answer as in Part 3: "I can use **chmod** to change the protections of files, and that will last after the hour is up." I accepted this, since I took off points for it in Part 3, and it's still a reasonable answer. The real answer is far more subtle. You want to basically copy **sh** to **plank**'s account and then do a **chmod 04755 sh**. But how do you make the copy? What you can do is take any file in **plank**'s directory except for **chmod**, and change its protection to 0666. Then you copy **/bin/sh** to that file — you can't use **cp** to do this, as it will delete the file first. So instead, you do:

```
UNIX> cat /bin/sh > file
UNIX> chmod 04755 file
```

Now you make your own copy of **/bin/sh** as in part 3 above, using **file** as a shell. A couple of you got close to this answer, but I accepted most reasonable attempts at answering it.



Question 2 Histogram

Question 3: 14 points

Part 1: The bug is that `time()` expects an argument, and procedure `b` uses `time()` without an argument. You might try to say that this is not a bug, but from the description of `time()` in the man page, it is clear that `time()` always uses its argument — when it says "if tloc is non-NULL, the return value is also stored in the location to which tloc points", that in no way implies that tloc cannot exist. What it says is that `time()` will check its argument to see if it equals NULL (which is zero). If so, it returns the time. Otherwise, it assumes `tloc` is a pointer, and sets its value to the time, and returns that value.

Some of you said that `b` doesn't have a type. In C, if a procedure doesn't have a type, it's assumed to be an `int`, so the declaration of `b` is ok. I gave you two points for that one, since it's reasonable that you didn't know that fact about C. All other "bugs" received no credit. Especially disappointing was the answer: "`i` is a temporary variable, so you can't return it". `i` is an `int` — of course you can return an `int`! We all do that all the time. If we returned a pointer to temporary memory, that is a problem, but `i` is not a pointer; it's just a plain old `int`.

Part 2:

```
    .globl t1

 b:
    add #-4, sp -> sp           / Allocate room for i

    ld [fp+16] -> %r0           / Push k on the stack
    st %r0 -> [sp]--

    ld [fp+12] -> %r0           / Push j on the stack
    st %r0 -> [sp]--

    jsr proc1                   / Call proc1

    add #8, sp -> sp            / Pop j and k off the stack

    st %r0 -> [fp]              / Store the return value of proc1 in i

    jsr time                    / Call time

    st %r0 -> t1                / Store the return value of time to
                                / global variable t1

    ld [fp] -> %r0              / Set the return value to i
    ret
```

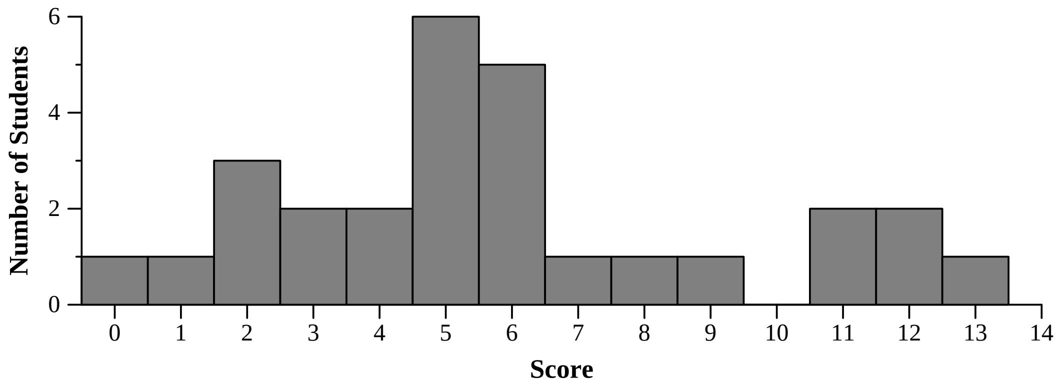
Part 3: Were there no bug, this code would return `proc1(j, k)` to the user, and set `t1` to the current time. However, there is a bug – that is that `time()` expects there to be an argument. It is going to look for that argument in `[fp+12]`, and what will be there? The value of `i`! So, if `i` is zero, then this code will run fine and won't need to be debugged. Otherwise, `time()` will try to treat `i` as a pointer and dereference it. This can have one of three results, depending on the value of `i`:

- If `i` is not a multiple of 4, you will get a bus error. This will be easy to find with a debugger.
- If `i` is not a legal address, you will get a segmentation violation. This will also be easy to find with a debugger.
- Otherwise, `time()` will overwrite some location in memory that it shouldn't be overwriting. This bug might go undetected, or it might be detected somewhere else in the program. It will be a nightmare

to debug — the best way to find this would be to use the program `lint`, which tries to find things like argument mismatches. In this case, `link` will actually say ``‘‘time:  variable # of args.’’``.

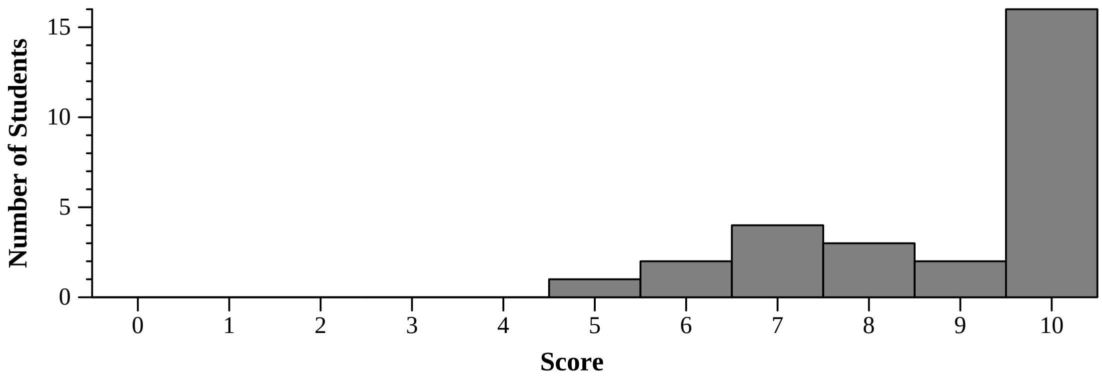
Grading: For part 2, grading was broken down into the following:

- 1 point for allocating room for `i` on the stack.
- 1 point for pushing `j` and `k` on the stack.
- 1 point for calling `jsr` for `proc1` and `time`.
- 1 point for popping `j` and `k` off the stack.
- 1 point for getting the return value of `proc1` into `i`
- 1 point for putting the return value into `r0`.
- Minus 1 point for changing `fp` or accessing the `pc`, or pushing the `fp` / `pc` on the stack.
- Half points for any of the above if you did them, but not really in the correct way.
- I rounded up in the end



Question 3 Histogram

Question 4: 10 Points



Question 4 Histogram

The answer to this was to traverse the directory, inserting files into a rb-tree keyed on the length of the filename. When that's done, you traverse the rb-tree, printing out the filenames, which are in the `v.val` field:

```
#include <dirent.h>
#include <stdio.h>
#include "rb.h"
#include <string.h>

main()
{
  DIR *dirp;
  struct dirent *d;
  Rb_node tmp, tree;
  char *namecpy;

  tree = make_rb();

  dirp = opendir(".");
  if (dirp == NULL) {
    perror("myls: .");
    exit(1);
  }

  for (d = readdir(dirp); d != NULL; d = readdir(dirp)) {
    namecpy = strdup(d->d_name);
    rb_inserti(tree, strlen(d->d_name), namecpy);
/* Or you can do: rb_inserti(tree, d->d_namlen, namecpy); */
  }
  closedir(dirp);

  rb_traverse(tmp, tree) {
    printf("%s\n", tmp->v.val);
  }
}
```

Note that you *must* do the `strdup()`. Otherwise, you may be setting the `v.val` fields to pointers which become invalid when then next `readdir()` or `closedir()` is called.

Grading:

- If you missed the `strdup()`, I took off one point.
- If you gave a good description instead of writing code, I gave you 8 points, although I took some off if the description was incomplete or incorrect.
- If you didn't use rb-trees, I took of at least 4 points. By this point, you should be familiar enough with rb-trees to be fluent with them, and to recognize when they should be used.