Changing execve's Function

Joel Sprinkle

December 4, 2015

Project for COMP 3000: Operating Systems

1 Introduction

1.1 The Idea

My original concept stems from seeing people get mad at their computer when they cannot make it do what they want it to do, or getting mad when their computer fails in some way, for example the famous blue screen of death. Seeing frustrated people swear at their computers sparked the idea in my mind that you should be able to call your computer names or bully the computer in order to get it to do what you want to. Expanding on this idea of calling your computer names, my idea accomplished this by changing the rename system call to stop moving/renaming any and all text files after 5 uses of the system call. After the initial 5 free passes the user had to append mean words to the end of the new filename in order to get the computer to move or rename a file, with meaner words providing a greater chance of having the rename system call complete the operation. A sample of this command is, $ mv /home/student/fileA.txt /home/student/temp/fileA-nerd.txt. Having the mean words appended to the end of the new file name not only bullied the system into completing the task, but it also left a lasting mark on the system, as well as the user because they are now stuck with the mean word appended to the end of their file, providing a little justice for the ”emotional scaring” suffered by the computer.

1.2 The Evolution

While my original idea worked, it was not very interesting as the mv command is not used very often, as well my idea only affected text files which made the scope of the hack even smaller and far less likely to be used. So I moved the base of my idea into the execve system call. Applying the same basic idea of having to call the system names or bully the system in order to get it to complete the system call. Now the student user only gets so many execve system calls before the student user has to bully the system in order for them to get it to do what they want. With the execve system call being used anytime a new process is launched, this morphed idea is a lot more in your face. With the student user only having 15 calls to execve, about half of which are taken up just by logging in, you have to start bullying the system quite quickly.

2 Design

By using the strace command it can be seen that the first thing that happens when executing an application is that the execve system call is run. All of my implementation happens in this execve system call, which can be found inside of the fs/exec.c file in the linux kernel source code[1]. On a high level my implementation works by interrupting the execve system call with my own code. This injected code checks to see which user is making the call, if it is the student user a counter is decremented by one allowing the system to perform as normal until the counter reaches zero. Once the counter reaches zero the student user is out of ”freebie” execve system calls, once this happens the code looks at the last argv element to see if it is one of the predefined ”bullying” words. If the last argv element is one of the words, the code generates a random number[2] between 1 and 100 providing an element of chance based on the specific bullying word that was entered in by the user, with meaner words providing a better chance of the operation completing. For instance, if you the user enters in, $ cd / Geek, the cd operation will be completed with a 70% chance. If the execve system call is to be completed, the ”Geek” argument is removed from the argv array before the completion of the execve system call so that the desired process executes properly, and the system leaves the user a response based on what was said to it in the kernel log. Or if the execve system call is not to be completed the system leaves a ”Better luck next time” message in the kernel log.

3 Implementation

3.1 Problems Encountered

One problem I encountered in implementing the hack was with the argv array. As I was quickly reminded using sizeof(argv)/sizeof(agrv[0]) does not work as a max number to iterate up to because argv has been passed to a function (in this case the execve system call), so it has lost some of its meta information. So both calls to the sizeof() function return 8 giving 8/8 = 1, which is useless. On trying different ways to iterate through the argv array I ended up making the system unbootable due to a what I am guessing was a seg-fault caused by a faulty loop. After that I did all my loop testing in a small C program I created, which proved to be less dangerous, easier, and quicker. I had to relearn how pointers and arrays get passed to functions in C, to overcome this problem[3][4].

Another problem I encountered using the argv array is that it is a const variable so it should not be changed, when I ignored this fact the compiler did not let the program compile. But I still needed to change argv to make the hack work. I could not get around this problem by changing the execve system call definition because any process using the execve system call is already expecting the current function definition. So I ended up solving this problem by assigning another variable to equal the argv array and then changing the new variable. The compiler still gives a warning when using the make bzImage command that the constant part of argv is being bypassed by assigning its value to another variable, but this is part of how the hack operates so special consideration was taken into account when altering the argv array, and the warning is being ignored.

4 Evaluation

Yes my code works. I know this through several different means of testing. The first way I tested the system was to login under the student account and keep running programs (i.e. ls, cd, nano, etc) until the system was out of freebee execve calls and then I observed how the system reacted. The system reacted by not letting any programs run, that is until I started adding the insults as a parameter to the program I wanted to run. Even then the programs did not execute every time, which was expected, due to each mean word only working a percentage of time, by the random number that is being generated. I could also see that the code was working by checking the ”come backs” to the mean words the system was generating in the kern.log file.

I also tested the stability of the system by building a kernel using the ”make bzImage” command. Using the student account to call the command completely fails, but the system does not crash, and is still stable and can continue to be used. Running the same command again using the root user account is successful and leaves the system in a stable state where it can continue to be used.

5 Conclusion

5.1 Contributions

The Kernel is a hard place to operate in. It is huge and full of code that I mostly do not understand, with that being said, it truly is amazing that it works at all. I

was able to add to the kernel though and produce a desired result. I changed the execve system call to refuse to do work or complete its operation without it being bullied into doing so. This change in the execve system call gives the system a little bit of a personality, the computer can say no to a user even though computers are designed to complete given instructions, making the system more interesting (or frustrating depending on your viewpoint) to interact with.

5.2 Limitations

One major limitation of the system is when the process that the student user executes, executes another secondary process, the secondary process always ends up failing. This failure happens because there is currently no way of daisy chaining the bullying of the system to the child processes of the process that was originally called. I noticed this limitation mostly when logging in or out and in the use of the shutdown command. If the student user is out of freebee execve calls the user is able to logoff (I am not sure if a proper logoff occurs though), but the student is then unable to log back into the system without a reboot of the system first. When attempting to log back into the student account, the system hangs on the ”Welcome to Ubuntu” screen for a while then returns to the login screen. Also when shutting down or rebooting the system, using the shutdown command, the system would hang and a forced hard reboot was necessary to get the system back online. However in testing the system I found this lack of being able to reboot the system easily quite annoying so I added an exception that looks for the shutdown command and lets the system shutdown properly.

5.3 Future Work

A couple of things could be done to the system to improve it. First removing the major limitation of not being able to daisy chain insults to child processes could be tackled, one way of overcoming this limitation could be to set an environment variable when the user insults the computer, this way the variable could then be passed down to and used in executing child processes. Another way the system could be improved upon would be to not only print out the systems comebacks in the kern.log file but to the current console window that is in use.

Notes

[1]Linux Cross Reference, http://lxr.free-electrons.com
[2]Linux Questions, http://www.linuxquestions.org/questions/programming-9/random-numbers-kernel-642087/
[3]Stack Overflow - passing an array of strings as a parameter, http://stackoverflow.com/questions/486075/passing-an-array-of-strings-as-parameter-to-a-function-in-c
[4]Stack Overflow - iterating through an array of strings, http://stackoverflow.com/questions/6812242/defining-and-iterating-through-array-of-strings-in-c

Appendix

The Code

All of the code that I added is inside of the execve system call, which is inside of the fs/exec.c file, with the exception of an include statement, #include <linux/random.h>, at the top of the exec.c file in order for the get_random_bytes() function to work. To put everything into context, I have included the the whole execve system call with my code starting on line 6 and ending on line 61.

```
1  SYSCALL_DEFINE3(execve,
2                  const char __user *, filename,
3                  const char __user *const __user *, argv,
4                  const char __user *const __user *, envp)
5  {
6      int cntr = 0, randNum;
7      static int num = 15, flag = 0;
8      char const **argPointer;
9      argPointer = argv; // causes a compiler warning, but I want to
           manipulate strings so have to -Playing with fire!
10
11     /* lets the system shutdown properly
12     if not included the system gets stuck
13     and a hard reboot is required */
14     if (strstr(filename, "shutdown")) {
15         flag = 1;
16     }
17
18     if (flag != 1) {
19         // generate random numbers from:
20         // http://www.linuxquestions.org/questions/programming-9/random
               -numbers-kernel-642087/
21         get_random_bytes(&randNum, sizeof(randNum));
22         randNum = randNum % 100;
23         if(randNum < 0) {randNum *= -1;}
24
25         // Troll only account where uid = 1000 (student) - protects the
               rest of the system
26         if(from_kuid_munged(current_user_ns(), current_uid()) == 1000)
               {
27             //printk(KERN_INFO "filename = %s, num = %d\n", filename,
                   num);
28             if(num == 0) {
29                 // look through the argv arguments for "bullying"
30                 while(*argPointer != NULL) {
```

```
31                      if (strstr(*argPointer, "Please") && (randNum < 3))
                            {
32                          *argPointer = NULL;
33                          argPointer = argPointer - cntr;
34                          printk(KERN_ALERT "execve: Ok I will comply,
                                since you asked nicely.\n");
35                          return do_execve(getname(filename), argPointer,
                                envp);
36                      } else if(strstr(*argPointer, "DOIT") && (randNum <
                            20)) {
37                          *argPointer = NULL;
38                          argPointer = argPointer - cntr;
39                          printk(KERN_ALERT "execve: Fine... No need to
                                yell.\n");
40                          return do_execve(getname(filename), argPointer,
                                envp);
41                      } else if(strstr(*argPointer, "Ass") && (randNum <
                            40)) {
42                          *argPointer = NULL;
43                          argPointer = argPointer - cntr;
44                          printk(KERN_ALERT "execve: Watch your tongue.\n
                                ");
45                          return do_execve(getname(filename), argPointer,
                                envp);
46                      } else if (strstr(*argPointer, "Geek") && (randNum
                            < 70)) {
47                          *argPointer = NULL;
48                          argPointer = argPointer - cntr;
49                          printk(KERN_ALERT "execve: That stings.\n");
50                          return do_execve(getname(filename), argPointer,
                                envp);
51                      }
52                      argPointer++;
53                      cntr++;
54                  }
55
56                  printk(KERN_ALERT "execve: Better luck next time.\n");
57                  return EACCES; //returns error
58              }
59              num--;
60          }
61      }
62
63      // original code
64      return do_execve(getname(filename), argv, envp);
65  }
```