Idea: You will be Forced to use NANO (Text Editor) Instead of VI:

Prepared by: Malik Ehsanullah (100833433)

I started off my project with a simple idea, “I want to change the functionality of “vi” command in shell, so that instead of opening a “Vim” text editor, it forces the user to work in nano instead.

For example when a user types:

vi foo.txt

Instead of opening a vim text editor, user is forced to work in nano.

So I took up this challenge and started my journey in the kernel land. I’m a newbie, so expect a lot of absurdity in my quest for modifying a simple vi command.

In order to do this miniscule hack, it was absolutely necessary that I try to pin down the exact system call that executes the “vi” or any command for that matter. So I started my journey with a simple

```
strace   -fqo   systemCallsLog.log  vi  new
```

Mind-bogglingly there were about 100s of system calls made, just to execute a simple vi command. I searched and searched, trying to figure out exactly which system call I should be targeting and dissecting to intercept the “vi” command and morph it into “nano”. It took me about 5-10 minutes to figure out that the system call at the very top of my log file had all the necessary tools I needed to tweak with the “vi” command.

```
execve(“usr/vin/vi”, [“vi”, “new”] , [/*20 vars */]) = 0
```

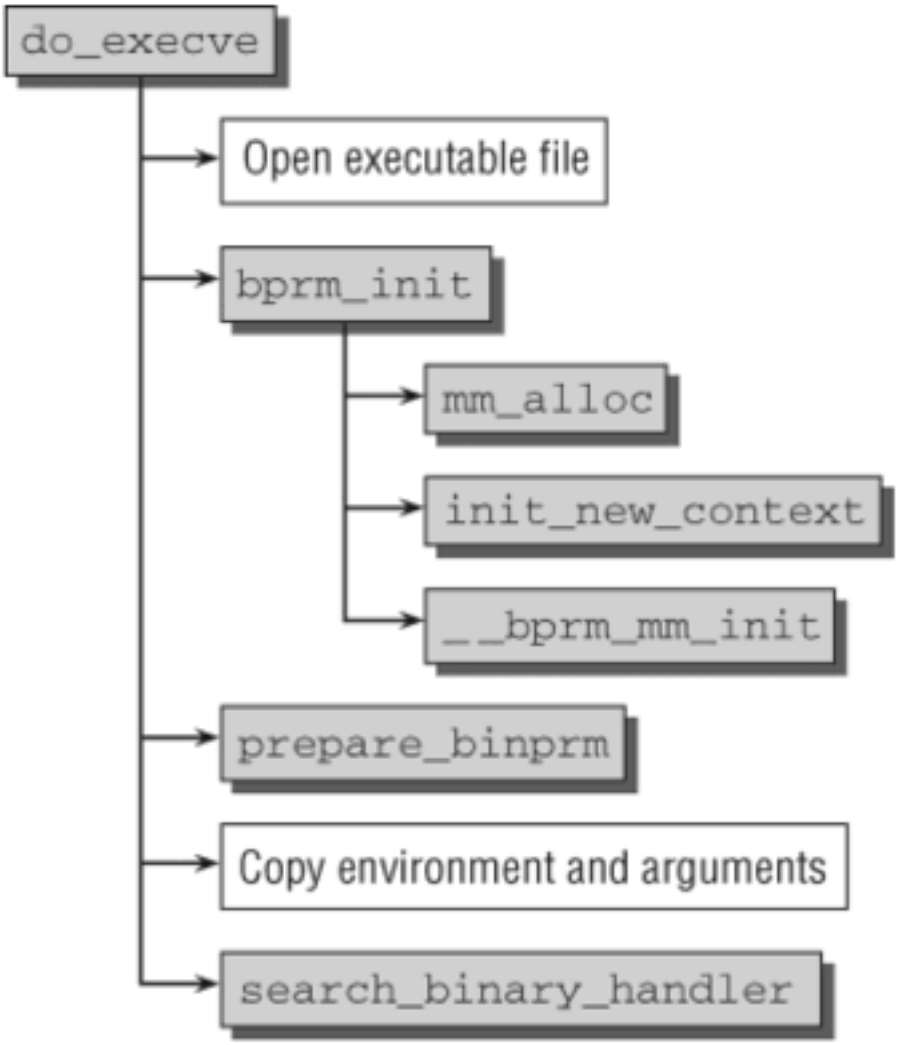
What if just change the arguments of the execve?

So I excitedly went on **lxr.free-electronic.com** and look up for execve. I found out that It is a function in **exec.c** file which is in the **fs** directory. Where exactly do I find that array, through which the string arguments are being passed?

Lets look at the break down of the functionality of do_execve, what it actually does and where I can intercept it.

Break down of do_execve:

- When execve is called, the first thing it does is it allocates memory space for linux_binprm. Linux_binprm is a data structure or a struct that is populated with the information about the new program that is going to be executed. Linux_binprm encapsulates in itself all the data required to execute a new program. It holds information about the new program's virtual memory address, the new executable file's name on the disk, its descriptor etc. linux_binprm is defined as a struct in source/include/linx/binfmts.h.
- Then "do_execve" does some checking for the memory and filename before it executes anything. In the lxr.free-eletrons you will find bunch of if statements that do all these checking. (I'm not exactly sure what they do, but they seem like check whether or not a program is executable and safe to run)
- Execve then checks the arguments and environment arguments that were passed into the function.
- It then seems to use "prepare_binprm" to fill the information in "linux_binprm" for which it allocated memory in the very first step. It initializes bunch of user permissions for the executable.
- It then copies filenames, command arguments and environment strings into the new executable program's page frame.



source: Linux Kernel Architect- By Wolfgang Mauerer

(Note: In the book, opening executable means, finding associated file's inode and generating file descriptor)

It feels like I should be adding code to change the environment and arguments.

After a brief search I found something that looked like it could be helpful.. Keyword, **filename.**

After a quick search on the lxr.free-electrons.com, I figured out that filename is a struct defined in the fs header file (fs.h). This header file can be found in **/linux-soruce/include/linux**

Over here a lot looked familiar and gave me something to work with. I found the filename struct defined as follows:

```
2079 struct filename {
2080        const char            *name;  /* pointer to actual string */
2081        const __user char      *uptr;  /* original userland pointer */
2082        struct audit_names      *aname;
2083        bool                  separate; /* should "name" be freed? */
2084 };
```

So seems like name and uptr are both pointers to filename.

So I started with dereferencing the “name” and “uptr” pointers in the exec.c and used a simple printk statement in the exec.c (in /linux-source/fs/exec.c)

```
printk(KERN_INFO” THIS IS A TEST  %s   %s”,  filename->name , filename->uptr);
```

The printk statement was placed on line 1484 of /source/fs/exec.c

```
1475        if (!bprm)
1476                goto out_files;
1477
1478        retval = prepare_bprm_creds(bprm);
1479        if (retval)
1480                goto out_free;
1481
1482        check_unsafe_exec(bprm);
1483        current->in_execve = 1;
1484  Code was added here printk(KERN_INFO” THIS IS A TEST %s  %s, filename-
>name, filename->uptr);
1485        file = do_open_execat(fd, filename, flags);
1486        retval = PTR_ERR(file);
1487        if (IS_ERR(file))
1488                goto out_unmark;
1489
1490        sched_exec();
```

Source: lxr.free-electroncs.com (linux-source, version 3.19) (source/fs/exec.c)

After doing : make bzImage

Sudo make install

Sudo shutdown -r now

And rebooting the OS I went into the tail /var/log/kern.log and found the following result.

```
Nov 23 14:45:08 localhost kernel: [ 5245.027740] THIS IS A TEST /bin/ls /bin/ls
Nov 23 14:45:13 localhost kernel: [ 5249.853204] THIS IS A TEST /usr/bin/vi /usr/bin/vi
Nov 23 14:58:09 localhost kernel: [ 6025.253230] THIS IS A TEST /usr/bin/tail
/usr/bin/tail
```

It seems like the arguments for “vi” are being passed through filename->name and filemname-> uptr.

I then proceeded to put an if statement right after printk

```
If (strcmp(filename->name,”/usr/bin/vi”) == 0  && strcmp(filename->uptr, ”/usr/bin/nano
)== 0 ){
filename-> name = “/usr/bin/nano”;
filename-> uptr = “/usr/bin/nano”;
}
```

This crashed the Kernel, for the reason unknown to me. Upon research and looking around I found that the “filename struct” holds all its attributes (including name and uptr) as constant, thus immutable.

The only way to change filename->name and filename->uptr is to make another pointer and point the filename->name pointer and filename->uptr pointers to it.

I then added a char pointer. (right after the if statement)

```
char string1 []  = “/usr/bin/nano
```

I then assigned string1 pointer to filename->name and filename->uptr.

After doing: make bzImage

Sudo make install

Sudo shutdown -r now

I restarted the kernel and typed "vi randomfilename" and nano was opened instead of vi and Wa-lah Success!

To summarize the changes I made: You just need to add the following lines. All these lines were added in exec.c in, /source/fs/exec.c

```
if ( strcmp(filename->name,"/usr/bin/vi") == 0
    && strcmp(filename->uptr,"/usr/bin/vi") == 0 )

printk(KERN_INFO " THIS IS A SUCCESSFUL CATCH------------  %s %s\n", filename->name, filename->uptr);
char string1[] =  "/usr/bin/nano";

filename->name = string1;
filename->uptr = string1;

 printk(KERN_INFO" TEST  %s........ %s", filename->name, filename->uptr);
}
```

in the lxr.free-electrons.com (linux version 3.19, in /source/fs/exec.c) I added the code on lines:

```
1475        if (!bprm)
1476                goto out_files;
1477
1478        retval = prepare_bprm_creds(bprm);
1479        if (retval)
1480                goto out_free;
1481
1482        check_unsafe_exec(bprm);
1483        current->in_execve = 1;
1484  Code was added here
1485        file = do_open_execat(fd, filename, flags);
1486        retval = PTR_ERR(file);
1487        if (IS_ERR(file))
1488                goto out_unmark;
1489
1490        sched_exec();
```

Conclusive Remarks:

1) Although I was able to change the exec.c to intercept a simple vi command, I haven't done much research on how to deal with bash commands that have extra arguments. For example, if you type vi –r filename.

2) Everything in the linux source code seems to be in form of struct.

3) When a process calls exec, the entire argv and environment arguments gets copied to the new process in the stack. Using the following line:

int execve(const char *filename, char *const argv[], char *const envp[]);

4) If a pointer is a const, you can't change its content but you can change the memory location it is pointing to.

5) In Kernel land there is no room for error, slightest mistake might cost you an hour or two to rebuilt and reconfigure the kernel.

6) You can either intercept bash commands using a module or you can edit the source code. Execve system call seems to be the first system call made by almost every command. So if you want to change functionality of a command, you should be able to intercept it in /source/fs/exec.c, more specifically (execve) .

References:

Mauerer, Wolfgang. *Professional Linux Kernel Architecture*. Indianapolis, IN: Wiley Pub., 2008. Print.

"Linux Cross Reference." *Linux/*. N.p., n.d. Web. 07 Dec. 2015.