CSE 509 Assignment 3: System-call and Library Interception

Akhil Bhutani

March 26, 2018

Warmup

(a) **strace - trace system calls and signals**

Usage -

$$strace \quad [options] \quad \underline{command} \quad [args]$$

Table 1: Syscalls for ls

read	write	open	close	fstat
mmap	mprotect	munmap	brk	rt_sigaction
rt_sigprocmask	ioctl	access	execve	getdents
statfs	arch_prctl	set_tid_address	openat	set_robust_list
prlimit64				

System calls made by ls can be listed using this command -

$$strace \ -c \ ls \ 2 > \&1 \ > \ /dev/null \mid cut \ -c52-70$$

The list of system calls is presented in Table 1.

System calls made by ls -l can be listed using this command -

$$strace \ -c \ ls \ -l \ 2 > \&1 \ > \ /dev/null \mid cut \ -c52-70$$

The list of system calls is presented in Table 2.

After running the command on both large and small directories, there is no difference in system calls made by ls command. On system calls made by ls -l command, we say more system calls as it queries all the files and directories for various attributes and properties and it prints a lot of information as compared to ls which lists only directory names. Hence, there is a surge in number of syscalls on running ls -l as compared to ls.

Table 2: Syscalls for ls -l

read	write	open	close	fstat
lstat	lseek	mmap	mprotect	munmap
brk	rt_sigaction	rt_sigprocmask	ioctl	access
socket	connect	execve	getdents	statfs
arch_prctl	getxattr	lgetxattr	futex	set_tid_address
openat	set_robust_list	prlimit64		

(b) **ltrace - a library call tracer**

Usage -

ltrace [*options*] *command* [*args...*]

System calls made by ls can be listed using this command

ltrace ls

We intend to filter out all utility function calls from the output. We can use the $-e$ option for this. The definition for $-e$ from man page is as follows :

$-efilter$ A qualifying expression which modifies which library calls to trace. The format of the filter expression is described in the section FILTER EXPRESSIONS. If more than one -e option appears on the command line, the library calls that match any of them are traced. If no -e is given, @MAIN is assumed as a default.

$$ltrace - c - e - @libc.so * -free - str * -get * \\ -fflush - fclose - mem * -realloc - file * \\ -fwrite * -opendir - readdir - bindtextdomain \\ - textdomain - ioctl - setlocale - closedir - isattyls$$

Using the above options will specify that we are not looking for any calls by libc.so library, other utility function calls like free, any string utility function, any memory related functions, locale functions and so on. Hence, we get only traces for calls that look like system calls. The $-c$ option lists only unique calls from the output. The output on executing the above command is :

Table 3: Output for calls that look like system calls

__ctype_get_mb_cur_max	__errno_location	__overflow
__freading	__fpending	__cxa_atexit
_setjmp		

(c) The following command is used to count the number of files accessed by applications when they are started up.

```
strace −f −e trace = file application_name 2 > &1 > /dev/null |
    cut −d ‘,’ −f 1,2 | cut −d ‘(’ −f 2 | cut −d ‘)’ −f 1 |
    sed ‘s/, \|,/\n/g’ | sed −n ‘/\//p’ | sort | uniq | wc − l
```

After running the above command with various application name, it will display number of unique lines which contains any file. The -f option is used to watch over all child processes and list the system calls used in them. For various applications, the number of files are -
ls : 14
nano : 82
libreoffice : 3387
firefox : 460
As we are required to list only number of files that are accessed, the above count lists only unique files generated using the above command. The count of files including the duplicates is:
ls : 20
nano : 137
libreoffice : 13724
firefox : 835

(d) The number of files that are loaded by each program can be listed using the following command:

```
strace −f −e trace = file application_name 2 > &1 > /dev/null |
grep”.so” | cut −d ‘,’ −f 1,2 | cut −d ‘(’ −f 2 | cut −d ‘)’ −f 1 |
    sed ‘s/, \|,/\n/g’ | sed −n ‘/\//p’ | sort | uniq | wc − l
```

For various applications, the number of files are -
ls : 9
nano : 9
libreoffice : 519
firefox : 185
As we are required to list only number of files that are loaded, the

above count lists only unique files generated using the above command. The count of files including the duplicates is:
ls : 14
nano : 14
libreoffice : 1395
firefox : 386

Tansparent Application Functionality Extension

I am writing third extension for this assignment which automatically creates backup copies of files before they are overwritten. Such a tool may be used to defend against ransom-ware.

The extension developed creates backup copies of files before they are overwritten. This can be very effective against ransom-wares. To avoid creating unnecessary backup copies, the backup is not created when files are opened in read-only mode.

The various system calls that can effect the file contents are -

1. **open** - Open files in various modes with various flags
2. **openat** - Open files in various modes with various flags
3. **creat** - Open a file in trunc mode if it already exists
4. **link** - Can be used while file is renamed or linked
5. **linkat** - Can be used while file is renamed or linked
6. **unlink** - Can be used when file is deleted
7. **unlinkat** - Can be used when file is deleted
8. **open_by_handle_at** - Obtain handle for a path-name and open file via a handle
9. **memfd_create** - Create anonymous file
10. **mknod** - create a special or ordinary file
11. **mknodat** - create a special or ordinary file relative to fd
12. **rename** - rename a file
13. **renameat** - rename a file relative to fd
14. **truncate** - truncate a file to a specified length

I am creating backup files for most commonly used system calls i.e. open, openat, creat, link, linkat, unlink, unlinkat. For all other system calls, the exploit/backup program code is similar and backup files needs to be created in the same way.

fflush system call is used when redirecting the output from stdout to a file. This can also be hooked eventually to save a backup if the file already exists. Currently it has not been implemented.

These set of system calls are used by many editors to create, rename, overwrite or delete a file.

Currently backups for regular files are created. However, if required backups for all files like symbolic links and other special files can also be created. All the system calls mentioned above are modified and object code is located in backupFiles.so to create a backup when file is opened in write-only mode, renamed or deleted. However, when file attributes are changed or permissions are altered we are not creating any copies. If we intend to create copies when the permissions are altered, we can use stat system call to first check for file permissions and then create a backup file with same permissions/attributes and then copy the contents. As this function call has much of its code duplicated to open where we create a backup file simply, this has not been added currently.

The backup directory is always located in $HOME as .backup folder. All backup files are appended with timestamps so as that the file names are unique and backup copies do not interfere with each other.

Currently, I have assumed $HOME directory contains the location of home directory and is not modified by malicious attacker. If the $HOME directory is left blank, we can simply check and place the backup directory in root. Other measures can be taken to make sure $HOME points to a valid user directory. As the backup folder is mostly stored in cloud, we can safely assume the above and ignore the condition checks for this assignment.

I have made sure the program works with Editors as well for bonus points. The underlying principle remains the same when working with editors. All editors use the same set of system calls to request writing to a file.

Execution

```
unset LD_PRELOAD
make
export LD_PRELOAD = /path_to_backupFiles.so/
```

README.md in the tar.gz includes the same information as above.