Strengthening Capsicum Capabilities with Libpreopen

by

© *Stanley Uche Godfrey*

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of *Science*

Department of *Scientific Computing*

Memorial University of Newfoundland

February 2018

St. John's Newfoundland

Abstract

The aim of the project is to develop a library called libpreopen to make it possible for application authors to sandbox their applications with capsicum without modification. At the moment, some applications that make OS system calls which Capsicum's compatible variants are implemented in libpreopen 's wrapper functions can be successfully sandboxed with Capsicum using libpreopen without any modification by the authors of these application.

When UNIX's commands cat, grep and wc are executed in FreeBSD shell and in Capsicum sandboxed capability mode with libpreopen, the cost of executing these UNIX's commands in Capsicum capability mode with libpreopen is almost the same as in executing them in FreeBSD.

Acknowledgements

I would like to express my gratitude to Dr. Jonathan Anderson for giving all the support and guidance I needed to complete the project. My thanks go to the Capsicum developer team and everyone who helped in reviewing the project and the project report.

Contents

List of Figures

Chapter 1

Introduction

1.1 Software application vulnerability

As more networks and gadgets are connected to the internet, the web becomes indispensable, it hosts productivity software suites for creating documents, spreadsheets and emails.Applications suites for scientific calculations, live television streaming and weather details are hosted on the web. Some of the services web applications provide include online banking, services for storing pictures and documents in the cloud. Software applications also provide services that connect home devices such IP cameras to mobile phones for remote monitoring and e-commerce services. Sensitive data like passwords and credit card details are usually required to access these services. Software applications, including ones that provide these services can have vulnerabilities which attackers can exploit, despite network defenses like firewalls and intrusion prevention systems [2].

A vulnerability in an application could be exploited by using a software attack technique known as Buffer Overflow. An application successfully attacked with Buffer overflow could crash often during execution, outputs different results from the expected result or hand total control of the system to the attacker. [3]

Heap Spraying is another type of software attack technique, a software successfully attacked with Heap Spraying executes the scripts of the attacker which carry out the intent of the attacker. Attacker intents are mostly to steal or corrupt data in a computer system. [1] Buffer over and Heap Spraying attacks are discussed further in the next section.

To mitigate attacks such a Buffer Overflow and Head Spraying, Capsicum an OS security system that executes applications in a sandboxed mode can be used. Executing an application is sandboxed mode means putting the application into compartments that are isolated from each other, other applications and users of the application. But for Capsicum to compartmentalize an application, the developer of the application must modify the application to conform to Capsicum sandboxing rules. Such modifications are rigorous and time-consuming and libpreopen relieves developers who want to make use of Capsicum to secure their applications of such application modifications

Chapter 2

Background

2.1 Buffer Overflow and Heap Spraying Attacks

A Buffer Overflow is a software attack technique used to exploit a programming bug in a non-memory safe program that results when a programmer fails to check if an input data is within the bounds of that input buffer. If the input data is more than the buffer can accommodate, the overflowing data will overwrite the contents of adjacent memory which may inject instructions to be executed into the process. An attacker who is able to add more data in a buffer than the buffer can accommodate could change the execution of applications, if the process is running with root user priviledge the attacker takes total control of the system.

The first known buffer overflow exploitation that gained mainstream media attention was accomplished by Robert Morris, a graduate student of Cornell University. [3] Morris wrote an experimental program that duplicates itself in a computer and disseminates itself to other computers through a computer network. Morris was able to put this program on the internet which was fast replicating and infecting computers. Morris' program, known as a *worm* exploited a buffer overflow bug in the UNIX Sendmail program, a program which runs on a computer and waits for connections from other computers which it receives emails from. Morris' program also exploited a buffer overflow in UNIX finger. UNIX finger is a program that prints out the login and other details of a logged in user in a UNIX system.

If an application running on either server or client side is vulnerable, attackers can use a technique known as Heap Spraying [1] to increase their chance of success by sending malicious code to the heap memory of the application in a computer. The Heap Spraying technique is used to duplicate the malicious code in different locations of the running application's heap memory to increase the chances of execution of the malicious code. Heap spraying is created with scripting languages like JavaScript. Different malware exploited vulnerabilities found in Internet Explorer 6 and 7 around 2004 when it was believed to be the most popular web browser.

These sorts of unauthorized access to computer resources by attackers are what Capsicum mitigates, and libpreopen will make Capsicum easier to use in limiting the damage intrusive malicious code from an attacker could cause.

2.2 Capsicum

Capsicum is a system that boosts UNIX security with sandboxed capability mode and capabilities. Capability mode is the ability of Capsicum to prohibit application compartments from interacting with each other except in a regulated manner using Capsicum capabilities rights. Application compartments (which are UNIX processes) in capability mode are totally isolated and are not allowed access to global namespaces such as file system namespace, process identifier (PID) namespace, interprocess communication namespace and socket-address namespace. The reason for these restrictions is to contain vulnerabilities to a compartment and not allow corruption to spread to other compartments of the application or the entire system.

Processes in total isolation cannot perform any task, this is where Capsicum capabilities are required. Capsicum capabilities are used to grant isolated processes in capability mode limited rights to perform specific actions in the capability token on a shared resource. For instance, a process may inherit file descriptors from a parent process or it may request access to a file from another process that has the right to send the file descriptor of the requested system file through IPC and before each of the processes enters capsicum capability mode. Regardless of how capability rights are acquired, processes in capability mode can only perform actions allowed in the capabilities granted on the file descriptors they acquire. A file descriptor acquired with capability right of CAP−READ cannot be used for fchmod(2) or have cap_write operations performed on it. [4]

An example of an application developed without Capsicum sandboxing in mind is tcpdump(1). tcpdump(1) is a command line application for printing protocols and packets transmitted or received over a connected network and for printing the communication of another user or computer. In a network through which unencrypted traffic such as telnet or HTTP passes, tcpdump(1) can be used by a superuser to view login details, URLs and the content of visited websites. Packet filters such as BPF can be used to limit the number of packets captured by tcpdump(1).[4] For tcpdump(1) to be sandboxed with Capsicum and have its privileges reduced, it must be modified with the code in listing (1.1) and (1.2) and analysed with a display utility such as procstat(1) tool to ensure that the capabilities exposed are the ones intended by the program author.

Listing 2.1: code to add capability mode to tcpdump

```
if (cap_enter() < 0)
 error("cap_enter: %s", pcap_strerror(errno));
```

Listing 2.2: code to narrow rights delegation in tcpdump

```
if (lc_limitfd(STDIN_FILENO, CAP_FSTAT) < 0)
 error("lc_limitfd: unable to limit STDIN_FILENO");
if (lc_limitfd(STDOUT_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
error("lc_limitfd: unable to limit STDOUT_FILENO");
 if (lc_limitfd(STDERR_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
 error("lc_limitfd: unable to limit STDERR_FILENO");
```

Chapter 3

Design and Implementation

3.1 Design

The design of libpreopen was made to strengthen Capsicum from these two viewpoints.

(1) libpreopen makes it possible for applications running in Capsicum capability mode to run some commands that require access to global namespaces via System calls without compromising the system security.

(2) libpreopen eradicates tedious application modifications developers have to make in order to use Capsicum compartmentalization and sandboxing to make their applications secure from attacks.

To achieve this design, libpreopen has its implementation of system calls open(2), access(2) and stat(2) at the moment. These functions are not allowed in Capsicum capability mode because access to global file namespace is required. Therefore libpreopen implements these system calls using the fstatat (2) ,openat(2) and faccessat (2) variants which are in harmony with Capsicum capability, making it possible for applications that make these system calls that require global file namespace access to be sandboxed in Capsicum without any modification. This is achieve in libpreopen by: (1) Preopening the directory of file resources needed by the application; (2) Store the directory descriptor and the absolute file path in an extendable storage; (3) Create a shared memory segment where the stored directory descriptors and their absoute file paths are mapped into.

When libpreopen is loaded with runtime linker ld_preload as environment variable, libpreopen is loaded before any other library and call to any of these system calls open(2), access(2) and stat(2) made in the environment will execute libpreopen's version of the function because the search for the function will first be made in libpreopen. Libpreopen replaces the pathname of system call the application is trying to make with corresponding directory descriptor and relative pathname and calls a variant of the system call that execute binary with file descriptor.

3.2 Implementation

To implement the features discussed in design section in libpreopen, the fields, structures and functions shown by the UML diagram in Figure 3.1 were written.

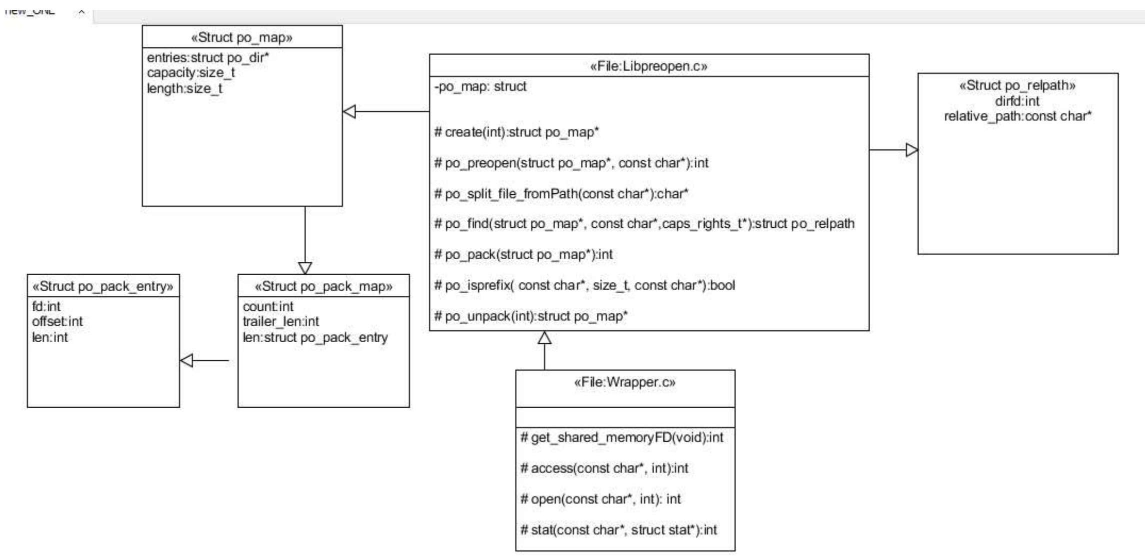

Figure 3.1: A UML diagram showing the relationships between fields, structures and functions in libpreopen

3.2.1 Create function

This function is called to create the extendable storage of libpreopen when the application incorporating libpreopen for Capsicum sandboxing is initialized. The create function has an integer parameter which it uses to set the initial capacity of the storage. The returned value of the create function is a pointer to the structure po_map.

3.2.2 po_preopen function

The po_preopen function takes pointer to po_map structure and a constant pointer to the data type char as arguments. The function then checks if the pointer to the constant char argument is a path to a directory or a path to a file. If it is a path to a system file, the function removes the file name from the constant char argument, call openat system call passing special value AT_FDCWD to ensure that the file descriptor of the current working directory is used, the constant char argument and O_DIRECTORY flag to make sure a directory file descriptor is returned by openat system call. Po_preopen function adds the directory descriptor returned by openat system call and the pointer to constant char argument to libpreopen extendable storage created when the application making use of libpreopen for Capsicum sandboxing was initialized. The function returns the directory descriptor returned by openat upon successful execution and 0 when its execution is unsuccessful.

3.2.3 Po_split_file_fromPath function

This function has a pointer to constant char parameter. It removes all the characters trailing a specified character, in this case, '\' from the point the first '\' is encountered from the tail of the parameter and returns this shorten character array.

3.2.4 Po_isprefix function

The function has three parameters, the path of a file system being requested by an application, the character length of this path and the paths stored in libpreopen's extendable storage which are iterated to see if any of the paths in libpreopen's storage is a prefix of the path to the file system being requested. The function returns true if a match is found and false otherwise.

3.2.5 Po_find function

This function makes use of po_isprefix function and Capsicum capabilities to search for a path in libpreopen's po_map storage which is a prefix of the pointer to the constant char parameter of the function. If a match is found and the operation requested by an application is allowed on the directory descriptor associated with this matching prefix in po_map by Capsicum capabilities. The relative path of the pointer to the constant char parameter of the function is returned. This relative path is extracted by moving the pointer to the constant char parameter to start from the location where the first '/' is encountered from the tail of parameter.

3.2.6 Po_pack function

This function takes a pointer to the po_map structure as its arguments. Creates a shared memory segment of the size of the number of elements in po_map extendable storage multiply by the size of po_pack_entry structure, plus the character length of each path in po_map. The function concatenates all the paths in the storage into a pointer to char data type. And stores the character length, offset of each path and the directory descriptor associated with each path in the po_map storage in an array of po_pack_entry structure which it puts in the shared memory segment. The function returns a descriptor to the created shared memory segment if it executes successfully and negative integer otherwise.

3.2.7 Po_unpack function

The po_unpack function access the shared memory segment with its integer parameter, creates a new array of po_map structure, unpacks the elements of po_pack_entry structure array in the shared memory segment into corresponding elements of an array of po_map structure and returns this array.

3.2.8 Wrapper functions

System calls such as open, access and stat that looks up binary by pathname are not allowed in Capsicum capability mode, because looking up binary by pathname requires access to global file system namespace. libpreopen provides wrapper function which are variants of the listed system calls that use file descriptor which can have Capsicum capability to execute binary.

At the moment libpreopen has its implementation of open, access and stat which it passes the directory descriptor and relative path returned by its po_find function to internally called openat, faccessat and fstatat function in its version of these system calls.

Chapter 4

Evalution

4.1 Functional

4.1.1 Capsh

Capsh is a shell program for compartmentalization and running of untrusted application in Capsicum capability sandbox. When a command is given to capsh to execute a program, capsh puts the name of the program to be executed and the program's arguments into a storage as shown in listing below.

Listing 4.1: The code snippet from `capsh's` `freebsd.c` file

```
 assert(argc > 0);


    vector<const string> args(argv, argv + argc);


assert(not args.empty());
```

capsh forks itself into a new process, enters Capsicum sandbox capability mode and sets file descriptors of shared libraries' directories as environment variables using LD_LIBRARY_PATH_FDS. Setting file descriptors of paths to library directories as environment variables is in line with Capsicum compartmentalization and sandboxing. After setting the environment variables, capsh starts the execution of the program with libc function fexecve(2).

Without libpreopen, capsh cannot do much sophisticated things. Only applications that do not require access to global file namespace can be executed on capsh, for example, echo(1), a command that prints strings on the terminals of UNIX shell programs. If an attempt to execute a command that requires access to global file system namespaces is made the 'action not permitted in capability mode' exception of Capsicum is thrown and the program execution terminated.

Capsh with libpreopen at the moment can execute some UNIX commands that require access to global file namespace for execution. Figure 4.1 shows how incorporation of libpreopen into capsh can make safe execution of an application that requires global file namespace in Capsicum possible.

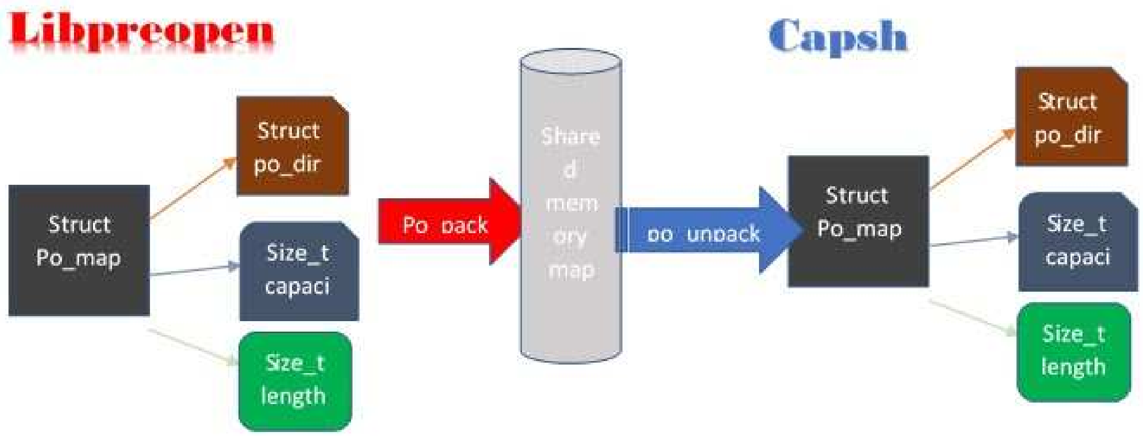

Figure 4.1: Shared memory mapping between libpreopen and a process started by capsh

From Figure 4.1, it can be observed that when libpreopen is used with capsh, capsh delegates pre-opening of directories an application to be executed in capsh may need access to their contents. The file descriptors and paths to these directories are put in an extendable storage, mapped into a shared memory segment and the file descriptor of the shared memory segment is returned to capsh by libpreopen. Capsh sets the returned shared memory segment's file descriptor as an environment variable, pre-loads libpreopen with ld_preload in the environment for the application capsh is about to start executing and fexecves to start the execution of the new application. During the execution of the application, if call to any of these system calls open(2) , access(2) and stat(2) are encountered, libpreopen's version of these functions which are Capsicum sandbox capability compatible will be called instead as described in section 3.2.8

Libpreopen was vetted by running some UNIX commands on FreeBSD shell, capsh without libpreopen and capsh with libpreopen. The UNIX shell commands used for vetting libpreopen are:

Echo which displays are given string in the terminal window. A look into the source code of echo , shows that no access to global file namespace is required to execute the command.

The ls command lists all files matching the name provided if no name is provided, ls list all files and directories in a directory. The source code of ls between lines 263 and 266 shows that a system call that requires access to a global file namespace is called as shown in the listing 3.1, libpreopen is yet to implement a Capsicum harmonious variant of this system call.

Listing 4.2: The code snippet for `UNIX command ls.c`

```
if ((ftsp =fts_open(argv, options,
    f_nosort ? NULL : mastercmp)) == NULL)
        err(1, NULL);
```

The command Head accesses global file namespace during its execution with fopen function which has not been implemented in libpreopen's wrapper functions. Listing 3.2 is head.c line 67-79 code.

Listing 4.3: The code snippet for file of `UNIX command head.c`

```
for (first = 1; *argv; ++argv) {
    if ((fp = fopen(*argv, "r")) == NULL) {
err(0, "%s: %s", *argv, strerror(errno));
            continue;
    }
```

```
}
```

The current libpreopen wrapper functions are able to make wc execute on file in Capsicum sandbox capability mode.

however command tail does not execute successfully because a function fopen and fstat will attemp access to global file namespace which is not allowed in Capsicum capability mode as shown in list 3.3, a code snippet of tail .c} between lines 138 and 142. Implementations of safe variants of these functions are yet to made in libpreopen

Listing 4.4: The code snippet for file of `UNIX command tail.c`

```c
if ((fp = fopen(fname, "r")) == NULL
    || fstat(fileno(fp), &sb)) {
            ierr();
            continue;
 }
```

The command grep executes successfully in capsh with libpreopn while strings command fails. Listing 3.4, a code snippet of strings .c between lines 114 and 119 shows that strings command makes a system call that accesses global file namespace. The Capsicum safe variant of this system call is yet to be implemented in libpreopen

Listing 4.5: The code snippet for file of `UNIX command strings.c`

```c
        if (!freopen(file, "r", stdin)) {
            (void)fprintf(stderr,"strings: %s: %s\n",
                file, strerror(errno));
             exitcode = 1;
```

```
            goto nextfile;
        }
```

4.2 Performance

To compare the performance oflibpreopen, The UNIX commands wc, grep and cat were executed on text files of sizes between 10MB and 1000MB, in a machine with the following specification, Intel (R) Xeon(R) CPU E3-1240 v5 @ 3.50GHz, 32 GiB RAM and UFS filesystem on a 7200 RPM disk.

Figure 4.2, is the graph of the time of execution of UNIX command wc in FreeBSD shell against the size of data the command wc is called on. And the execution time of wc with libpreopen in capsh against the size of data wc is called on.

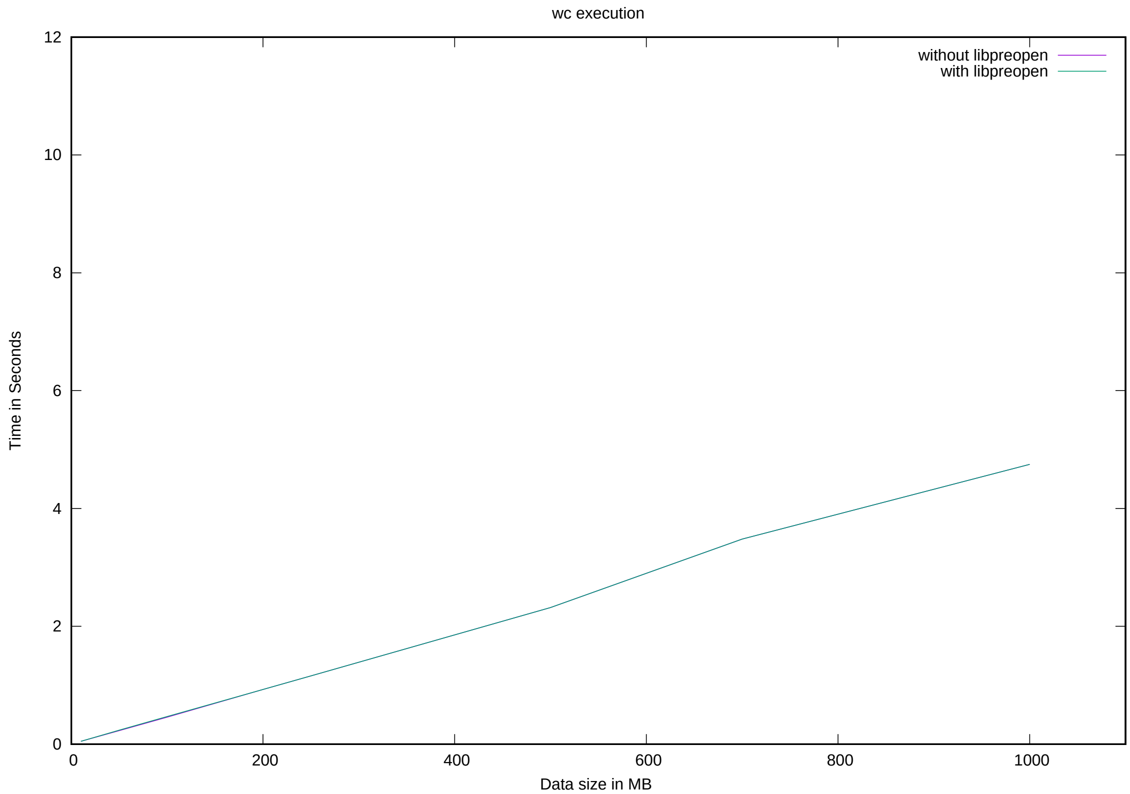

Figure 4.2: Data size vs time graph of wc command in FreeBSD shell and capsh

From figure 4.2, it can be seen that cost of running wc command in both FreeBSD shell, and capsh is the same on the same data size.

grep is a UNIX command that executes successfully in capsh with libpreopen. Figure 4.3 compares the cost of running grep command on text files of size between 10 MB and 1000 MB in FreeBSD shell and capsh.

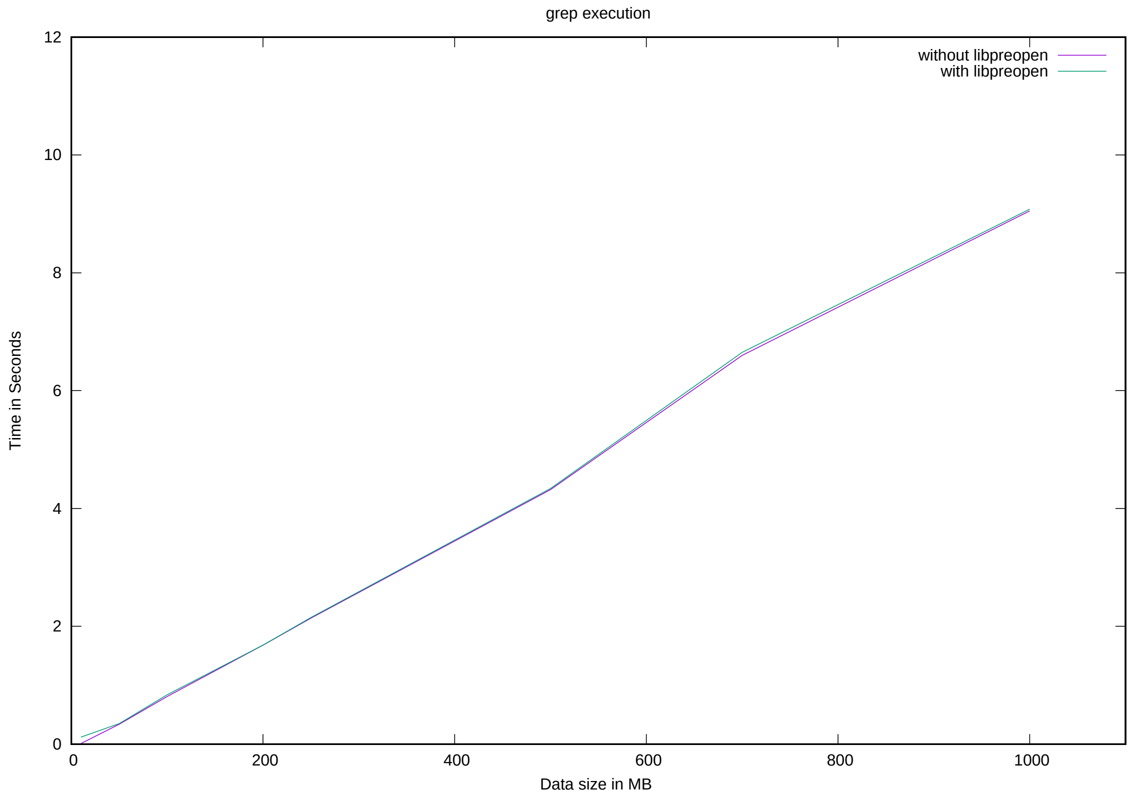

Figure 4.3: Data size vs time graph of grep command in FreeBSD shell and capsh

From Figure 4.3 it can be seen that the performance of grep command in FreeBSD shell, and capsh are almost the same.

The third FreeBSD command that execute successfully in capsh at the moment, is cat. Cat displays the content of file and the larger the file the longer the execution duration. figure 4.4 shows the performance of cat in FreeBSD shell and casph.

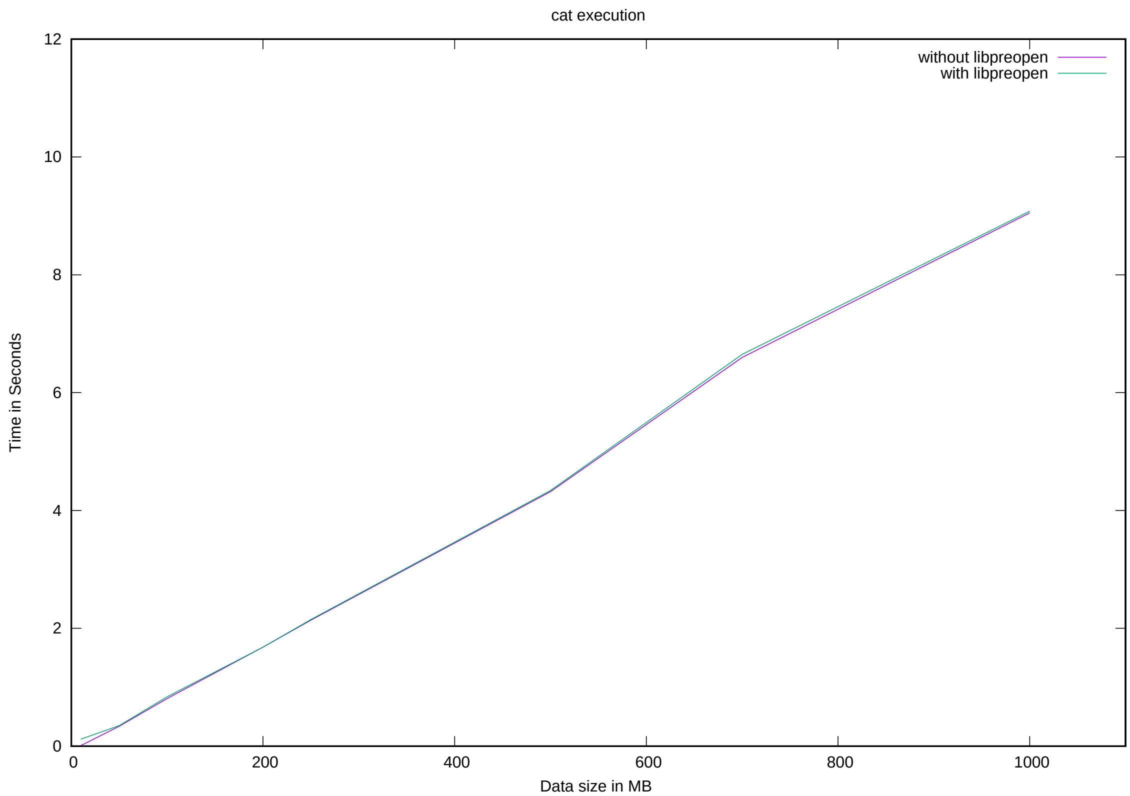

Figure 4.4: Data size vs time graph of cat command in FreeBSD shell and capsh

figure 4.4 shows that the cost of executing cat on a file in FreeBSD shell is almost the same as the cost executing cat on a file of same size in capsh with libpreopen

4.3 Future work

Not all UNIX commands are able to execute successfully in capsh with libpreopen. Such UNIX commands include head, tail, strings, and ls. These UNIX commands call variants of libc functions which are not allowed in Capsicum sandbox capability mode and which Capsicum compatible variants are yet to be implemented in libpreopen wrapper functions. Such libc functions include fopen, freopen , fstat and others yet to be identified. Including Capsicum harmonious variants of these functions will make it possible for more applications to be sandboxed with Capsicum using libpreopen without the authors of the applications making modifcations to make their applications conform to Capsicum sandboxing rules.

Chapter 5

Conclusion

If a vulnerability in an application is exploited and malicious data injected into the system, Capsicum mitigates the spread of the malicious data by con

fining them to the affected process, since an application running in Capsicum sand- boxed mode is compartmentalized into processes and each process sandboxed.

However, an application running in Capsicum sandboxed capability mode is forbidden to access global OS namespaces such as file system, process IDs, IPC namespaces. The application also has a restricted access to system calls while access to system calls that involves global namespace access is forbidden. For Capsicum to contain the damage exploit of a vulnerability in an application can cause, the application has to give up its right to perform certain operations.

Applications running in Capsicum capability mode can acquire Capsicum capability rights, and libpreopen makes it possible for such applications to request system call operations which the applications have the Capsicum capability rights for and have libpreopen performs these system call operations with libpreopen's version of these system calls.

The cost of sandboxing an application with Capsicum using libpreopen is insignificant compare to hours of rigorous application modification by applicaitons authors in order to make their application conform to Capsicum sandboxing rules. Application authors wishing to run unmodified binaries in a sandboxed Capsicum capability mode should use libpreopen libary in order to avoid tedious and time consuming application modification.

Bibliography

[1] A. Ansari. Heap spraying. https://www.exploit-db.com/docs/english/31019-heap-sprayi
Accessed December 16, 2017.

[2] P. Lonescu. The 10 most common application attacks in action. https://securityintelligence.com/the-10-most-common-application-attacks-in-act
Accessed December 15, 2017.

[3] Veracode. What is a buffer overflow learn about buffer overrun vulnerabilities exploits and attacks. https://www.veracode.com/security/buffer-overflow/. Accessed December 15, 2017.

[4] R. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum:practical capabilities for unix., 2011.

Appendix A

Appendix title

This is Appendix A.

You can have additional appendices too (*e.g.*, `apdxb.tex`, `apdxc.tex`, *etc.*). If you don't need any appendices, delete the appendix related lines from `thesis.tex` and the file names from `Makefile`.