Strengthening Capsicum Capabilities with Libpreopen

by

© *Stanley Uche Godfrey*

A report 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

March 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 for which Capsicum's compatible variants are implemented in wrapper functions of libpreopen 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 executing them in FreeBSD shell without Capsicum.

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

Abstract **ii**

Acknowledgements **iii**

List of Figures vi

1 Introduction **1**

1.1 Software application vulnerabilities 1

2 Background **3**

2.1 Buffer Overflow and Heap Spraying Attacks 3

2.2 Capsicum 5

3 Design and Implementation **7**

3.1 Design 7

3.2 Implementation 8

3.2.1 po_map_create function 9

3.2.2 po_split_file_frompath function 9

3.2.3 po_isprefix function 10

3.2.4 po_find function 10

3.2.5 po_pack function 11
3.2.6 po_unpack function 11
3.2.7 po_preopen function 12
3.2.8 Wrapper functions 12

4 Evaluation **14**

4.1 Functional evaluation 14
4.2 Performance 19

5 Conclusion **24**

5.1 Future work 25

Bibliography **27**

List of Figures

3.1 libpreopen UML 9

4.1 Shared Memory Mapping 16

4.2 libpreopen's vetting outcome 19

4.3 wc's Datasize vs Time-diff graph 20

4.4 grep's Datasize vs Time-diff graph 21

4.5 cat's Datasize vs Time-diff graph 22

Chapter 1

Introduction

1.1 Software application vulnerabilities

As more gadgets are connected to networks and networks are connected to each other, vulnerable programs are exposed to malicious attackers. Examples of such programs are software suites for creating documents, spreadsheets, emails, scientific calculations, live television streaming, weather details, online banking, services for storing pictures and documents in the cloud, 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 [5].

A vulnerability in a software application could be exploited by using a software attack technique known as a buffer overflow. An application successfully attacked

with a buffer overflow could crash during execution, output different results from the expected result or hand total control of the system to the attacker [8].

Heap Spraying is another type of software attack technique. 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 [3]. Buffer overflow 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 in a sandboxed mode means putting the application into compartments that are isolated from each other, other applications and users of the applications. Capsicum abates the damage an attacker can do to an application by compartmentalizing the application and with libpreopen, running an application in Capsicum sandboxed mode to alleviate the damage of intrusive code from an attacker is even easier.

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 [8]. 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 privilege 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 [8]. 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 [3] 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 [7] 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. Application compartments (which are UNIX processes) in capability mode are totally isolated and are not allowed access to global namespaces such as the file system namespace, process identifier (PID) namespace, interprocess communication namespace or socket-address namespace [2]. 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 Inter-process Communication (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 [13].

An example of an application developed without Capsicum sandboxing in mind is tcpdump(1) [6]. 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). For tcpdump(1) to be sandboxed with Capsicum and have its privileges reduced, it must be modified with the code in listing (2.1) and (2.2) 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

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

Listing 2.2: code to narrow rights delegation in tcpdump

```
1 if (lc_limitfd(STDIN_FILENO, CAP_FSTAT) < 0)
2  error("lc_limitfd: unable to limit STDIN_FILENO");
3 if (lc_limitfd(STDOUT_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
4 error("lc_limitfd: unable to limit STDOUT_FILENO");
5  if (lc_limitfd(STDERR_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
6  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 the following viewpoints.

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

(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 own implementation of the 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 capabilities, making it possible for applications that make these system calls that require file namespace access to be sandboxed in Capsicum without any modification. This is achieved in libpreopen by: (1) preopening the directory of file resources needed by the application; (2) storing the directory descriptor and the absolute file path in an extendable storage; (3) creating a shared memory segment where the stored directory descriptors and their absoute file paths are mapped into for sharing across process boundaries.

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

3.2 Implementation

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

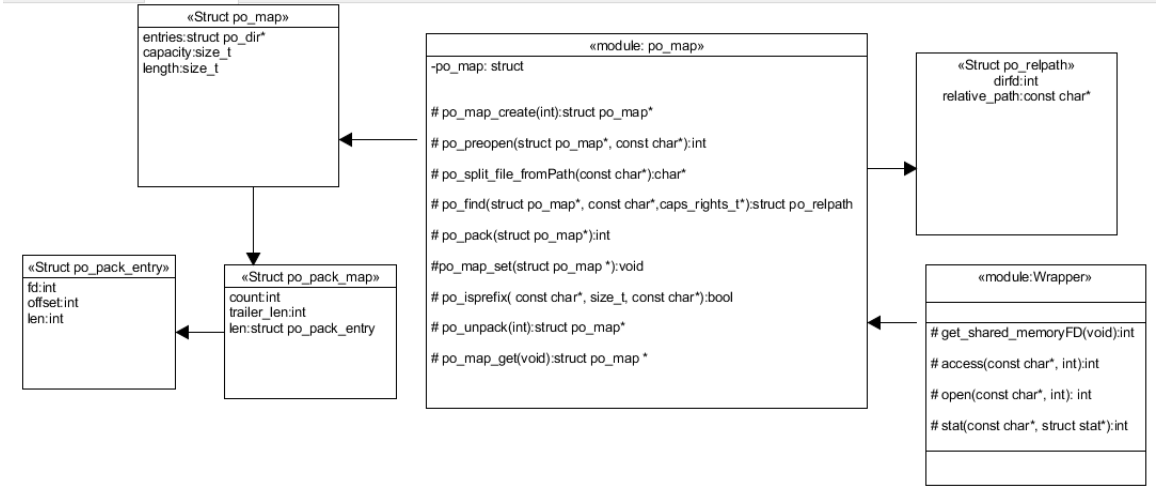

Figure 3.1: A UML diagram showing the relationships between fields, structures and functions in libpreopen

3.2.1 po_map_create function

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

3.2.2 po_split_file_frompath function

The purpose of this function is to seperate the file name from the absolute path passed as argument. It has a pointer to a constant string 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 shortened character

array.

3.2.3 po_isprefix function

The function determines if any of the directory paths in the extendable storage po_map, is a prefix of the absolute path to a file given. It 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 absolute path to the file system given. The function returns true if a match is found and false otherwise. For example if a directory with absolute path /usr/somedir/anotherdir/ has been pre-opened and an application is requesting a file with absolute path /usr/somedir/anotherdir/file, po_isprefix function returns true.

3.2.4 po_find function

This function finds a directory which is a prefix to a given absolute path to a system file and which has the Capsicum capability rights required. It makes use of po_isprefix function and Capsicum capabilities to search for a path in libpreopen's po_map. If a match is found, 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.

3.2.5 po_pack function

This function packs the extendable storage po_map into an inheritable form that can be put in a shared memory segment; making it possible for po_map to be inherited across process execution boundary. The function takes a pointer to the po_map structure as its arguments and creates a shared memory segment of the size of the number of elements in po_map extendable storage multiplied by the size of po_pack_entry structure, plus the character length of each path in po_map. Because string pointer cannot be passed from one process to another but can only be copied as a block of data, the function concatenates all the paths in the storage into a pointer to char data type, 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 as a packed-in buffer representation of po_map. The function then puts this packed-in buffer representation in the shared memory segment and returns a descriptor to the created shared memory segment if it executes successfully and a negative integer otherwise.

3.2.6 po_unpack function

The po_unpack function accesses a shared memory segment with its integer file descriptor (FD) 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.

3.2.7 po_preopen function

The po_preopen function takes a pointer to a 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, calls the openat(2), passing AT_FDCWD to ensure that the file descriptor of the current working directory is used, the constant char argument and the O_DIRECTORY flag to make sure a directory file descriptor is returned by openat(2). The po_preopen function adds the directory descriptor returned by openat(2) and the pointer to constant char argument to libpreopen's extendable storage created when the application making use of libpreopen for Capsicum sandboxing was initialized. The function returns the directory descriptor returned by openat(2) upon successful execution and -1 when its execution is unsuccessful. The execution will be unsuccessful if the given path is not a path to a system file.

3.2.8 Wrapper functions

System calls such as open(2), access(2) and stat(2) that look up files by pathname are not allowed in Capsicum capability mode, because looking up files by pathname requires access to the global file system namespace. libpreopen provides wrapper functions 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(2), access(2) and stat(2) which passes the directory descriptor and relative path returned by its po_find func-

tion to open(2), access(2) and stat(2) when the parent directory has been pre-opened.

Chapter 4

Evaluation

To ascertain that libpreopen can be used to run an application in Capsicum capability mode without modifying the application to conform to Capsicum sandboxing rules. And to measure the cost in time of using libpreopen to sandbox the application against the cost in time of running the application without sandboxing, some UNIX commands were used. These commands which were executed in a Capsicum capability aware program capsh are echo, ls, less, head, wc, tail, grep, strings, and cat.

4.1 Functional evaluation

Capsh is a capability aware shell for running untrusted application in Capsicum sandboxes [1]. 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 an argument vector, forks itself into a new process, enters a Capsicum sandbox capability mode and passes the file descriptors of shared libraries' directories as environment variables using LD_LIBRARY_PATH_FDS. Setting file descriptors of paths to library direc-

tories as environment variables is in line with Capsicum compartmentalization and sandboxing. Because environment variables persist across fexecve(2), capsh starts the execution of the program with the system call fexecve(2).

Without libpreopen, capsh cannot do very sophisticated things. Only applications that do not require access to the 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" error occurs, after which the program execution typically terminates.

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

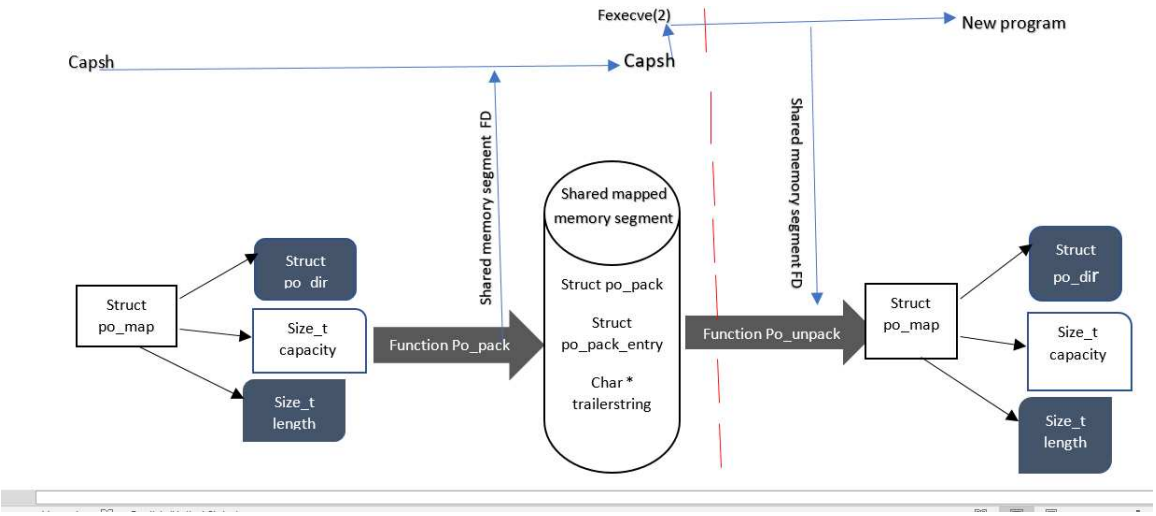

Figure 4.1: Shared memory segment mapping between libpreopen and a process started by capsh

From Figure 4.1, it can be observed that when libpreopen is used with capsh, capsh pre-opens the directories that an application to be executed in capsh may need. These directories are passed as arguments to capsh and capsh uses libpreopen's po_preopen function to ascertain if there is a path in libpreopen's extendable storage that is a prefix to any of the arguments. libpreopen has an extendable storage which is a pointer to libpreopen's structure po_map and stores already-opened directories' path and their descriptors. If a prefix is found, the directory descriptor of the prefix and the characters of the argument after those matching the prefix are used as arguments for libpreopen's wrapper functions. If no prefix is found for any of the arguments passed to capsh, libpreopen opens the directory and stores both the directory's path and its descriptor in the pointer to po_map structure. This pointer is put into a shared memory segment created by libpreopen and capsh sets the returned shared

memory segment's file descriptor as an environment variable, pre-loads libpreopen with LD_PRELOAD in the environment and calls fexecve(2) to start the execution of the new application. During the execution of the application, if calls 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−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(1), ls(1), grep(1), head(1), tail(1), wc(1) and strings(1)

echo(1) displays given string in the terminal window. A look into the source code [4] shows that no access to the global file namespace is required to execute the command.

The ls(1) command lists all files matching the name provided; if no name is provided, ls(1) list all files and directories in a directory. The source code of ls(1) [10] 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 4.1. libpreopen is yet to implement a Capsicum−harmonious variant of this system call.

Listing 4.1: The code snippet for `UNIX command ls.c`

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

The command head(1) has functions that may access the global file namespace during their execution and have not been implemented in libpreopen's wrapper func-

tions as shown by its source code [9].

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

However the command tail (1) does not execute successfully because its source code [11] makes some system calls that attempt access to the global file namespace which are not allowed in Capsicum capability-safe mode and are yet to be implemented in libpreopen's wrapper functions.

The command grep executes successfully in capsh while the strings (1) command fails. Listing 4.2, a code snippet of strings [12] between lines 114 and 119 shows that the strings command makes a system call that accesses the global file system namespace. The Capsicum-safe variant of this system call is yet to be implemented in libpreopen.

Listing 4.2: The code snippet for file of `UNIX command strings.c`

```
1         if (!freopen(file, "r", stdin)) {
2             (void)fprintf(stderr,"strings: %s: %s\n",
3                 file, strerror(errno));
4              exitcode = 1;
5              goto nextfile;
6         }
```

After vetting libpreopen by runing UNIX shell commands Echo(1), ls (1), grep(1), head(1), tail (1), wc(1), less (1) and strings (1) on FreeBSD shell, capsh without libpreopen and capsh with libpreopen, the outcome of the vetting process is shown in Figure 4.2.

Commands	Works with Libpreopen	Does not work with Libpreopen	Works on the shell
echo	*		*
ls		*	*
less	• Works with warnings		*
head		*	*
wc	*		*
tail		*	*
grep	*		*
strings		*	*
cat	*		*

Figure 4.2: The outcome of libpreopen's vetting with some UNIX shell commands

From Figure 4.2, it can be observed that applications which make systems calls whose Capsicum capability-safe variant has been implemented as a libpreopen's wrapper function can be susccessfully sandboxed with Capsicum incorporating libpreopen wihout modifying the applicaiton to conform to Capsicum sandboxing rule .

4.2 Performance

To measure the performance of libpreopen, The UNIX command time was executed on the following UNIX commands wc, grep and cat 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 4.3 is the graph of the the percentage difference of time in seconds of the execution of UNIX command wc in FreeBSD shell and in capsh incorporating libpreopen against the size of data wc is called on.

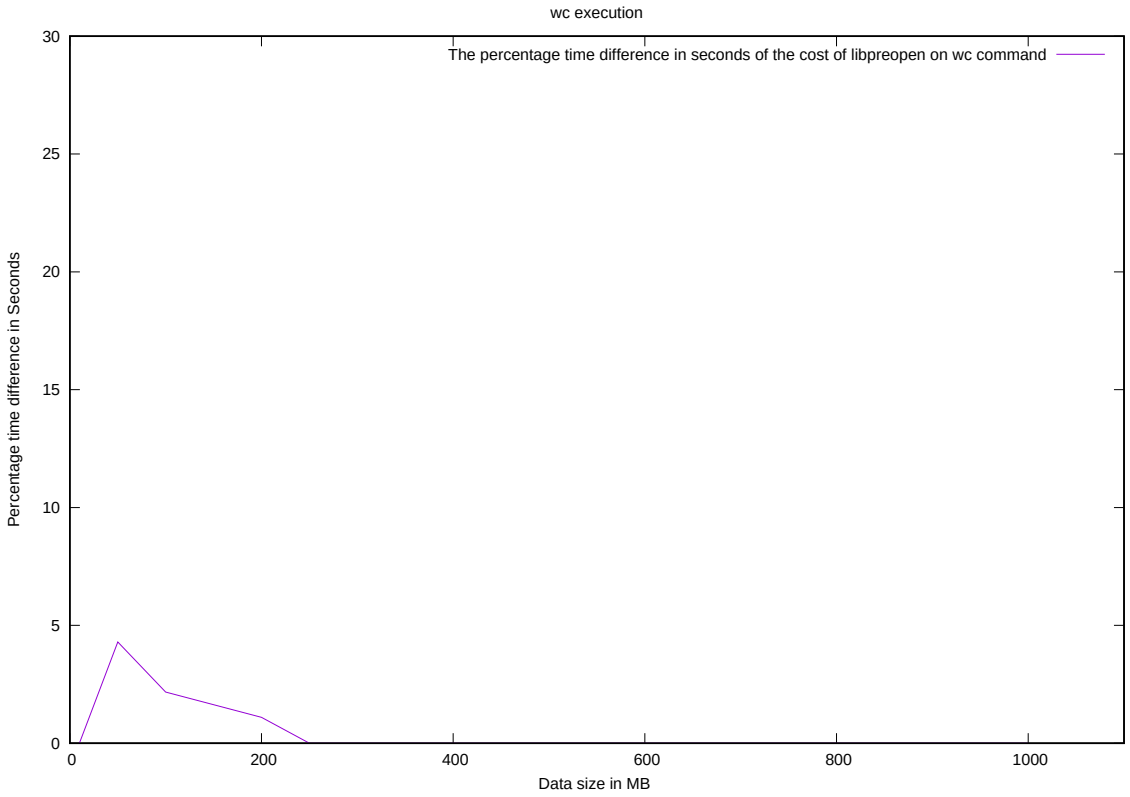

Figure 4.3: Data size vs percentage time-difference graph of wc command in FreeBSD shell and capsh

From Figure 4.3, it can be seen that data size vs time-difference percentage cost of running wc command in capsh incorporating libpreopen vs running wc command in FreeBSD shell increases sharply for data size of 0 MB to 40 MB, and peaks at 4%

when data size is about 40 MB, probably because of extra time required to pre-open system file paths by libpreopen. Above 40 MB of data size, the percentage cost of running wc in libpreopen starts falling sharply and becomes equal to that of FreeBSD shell at data size of about 250 MB.

grep is a UNIX command that executes successfully in capsh with libpreopen. Figure 4.4 measures the data size vs percentage time-difference cost of running grep command on text files of size between 10 MB and 1000 MB in FreeBSD shell and in capsh with libpreopen.

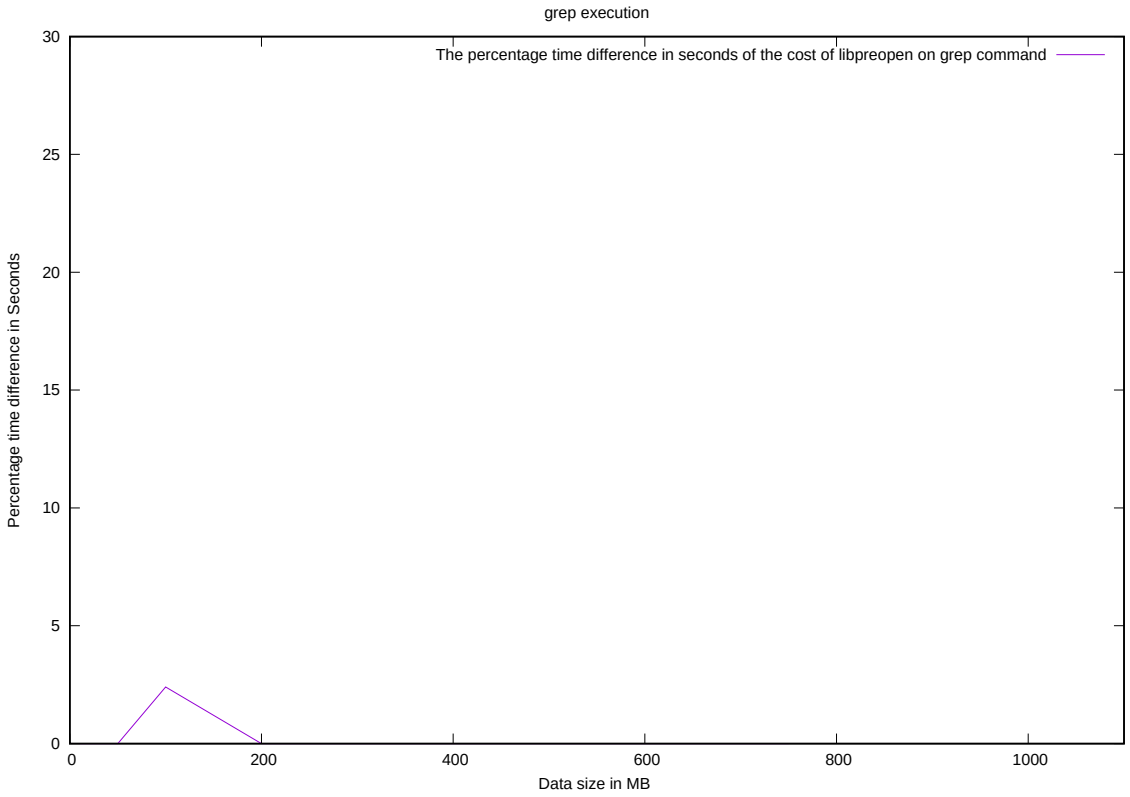

Figure 4.4: Data size vs percentage time-difference graph of grep command in FreeBSD shell and capsh

Figure 4.4 is the measure of data size vs percentage time-difference of the cost of running grep command in FreeBSD shell, and in capsh with libpreopen. The latter lags with about 2.5% difference in cost when data size is between 0 MB and 80 MB. The performance is the same for data size of between 200 MB and above.

The third UNIX command that executes 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.

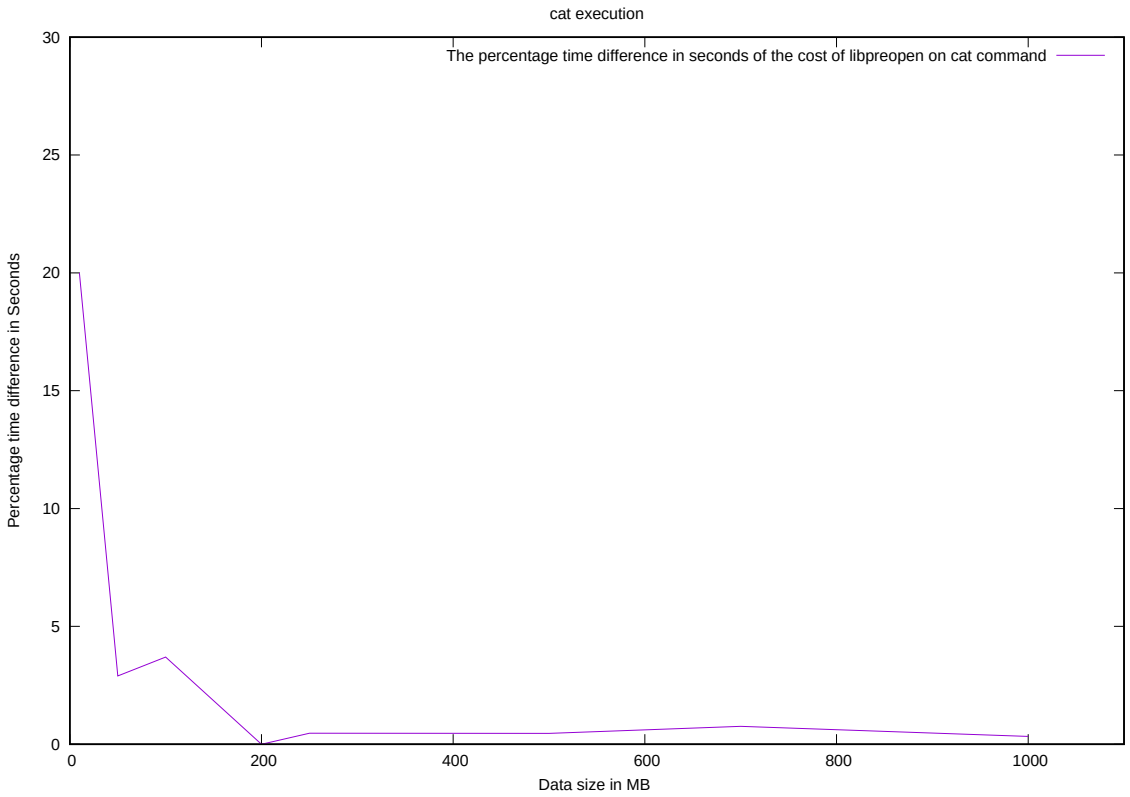

Figure 4.5: Data size vs percentage time-difference graph of cat command in FreeBSD shell and in capsh with libpreopen

Figure 4.5 shows that the time-difference cost of executing cat on a file in FreeBSD shell is 20% less, when the data size is between 0 MB to 50 MB, about 4% less when the data size is between 50 MB and 120 MB, falls gradually from data size of 120 MB and becomes the same as executing the command in capsh with libpreopen when the data size is about 200 MB. When data size is more than 200 MB, executing the command in capsh with libpreopen cost about 1% extra of the time it cost to execute the command in FreeBSD shell and execution of the command costs the same in both for data size above 1 GB.

Vetting proves that with libpreopen, the cost of dealing with pre−opening is amortized over larger executions, for example, with the UNIX command wc, the cost of execution of the command on a text file of 10 MB is 0%, 4% for a text file of 50 MB, 2.17% for 100 MB of text file, 1.1% for a text file of 200 MB and 0% for text files of 250 MB and above. The cost of dealing with pre−opening is also amortized over larger executions of the UNIX commands cat and grep.

Chapter 5

Conclusion

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

If a vulnerability in an application is exploited and malicious data injected into the system, Capsicum mitigates the spread of the malicious data by making sure that they are confined in the corrupt process, since an application running in Capsicum sandboxed mode is compartmentalized into processes and each process sandboxed.

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 perform these system call operations with libpreopen's version of these system calls.

After evaluating the effectiveness of libpreopen by executing UNIX commands wc, grep and cat in FreeBSD shell and in capsh with libpreopen, it shows that the cost of running these applications in Capsicum sandbox mode is less than 5% of the cost of running the applications without Capsicum sandboxing and decreases as file size increases. This cost is insignificant compare to hours of rigorous application modification by authors in order to make their applications conform to Capsicum sandboxing rules. Application authors wishing to run unmodified binaries in Capsicum capability mode to secure their applications from attacks and limit damage done to computer resources by an attacker if in the worst case scenario the attacker succeeds should use libpreopen for fast, easy and cost effective Capsicum sandboxing.

5.1 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's wrapper functions. Such libc functions are yet to be identified and including Capsicum harmonious variants of these functions when identified 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 applica-

tions conform to Capsicum sandboxing rules.

Making libpreopen an integral part of FreeBSD's libraries is another work to be completed in the future, to avoid seperate compilation and installation of libpreopen's source code in FreeBSD's before it can be used by developers for Capsicum sandboxing.

Bibliography

[1] J. Anderson and S. U. Godfrey. Capsh. https://github.com/musec/capsh. Accessed March 1, 2018.

[2] J. Anderson, S. U. Godfrey, and R. Watson. Towards oblivious sandboxing with capsicum. *FreeBSD Journal*, Proceedings of the 2017 BSDCAN USENIX(20), 2017.

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

[4] FreeBSD. echo source code. https://github.com/freebsd/freebsd/blob/master/bin/echo Accessed December 15, 2017.

[5] 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.

[6] F. M. Pages. tcpdump(1). https://www.freebsd.org/cgi/man.cgi?tcpdump(1). Accessed March 1, 2018.

[7] A. Tu. Exploiting the microsoft internet explorer malformed iframe vulnerability. https://pen-testing.sans.org/resources/papers/gcih/exploiting-microsoft-intern 2004. Accessed March 1, 2018.

[8] 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.

[9] ViewVC. head.c source code. https://svnweb.freebsd.org/csrg/usr.bin/head/head.c?vi Accessed December 15, 2017.

[10] ViewVC. ls source code. https://svnweb.freebsd.org/csrg/bin/ls/ls.c?revision=67594 Accessed December 15, 2017.

[11] ViewVC. tail.c source code. https://svnweb.freebsd.org/csrg/usr.bin/tail/tail.c?vie Accessed December 15, 2017.

[12] ViewVC. tail.c source code. https://svnweb.freebsd.org/csrg/usr.bin/strings/strings Accessed December 15, 2017.

[13] R. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: practical capabilities for unix. *Proceedings of the 19th USENIX Security Symposium*, (3), 2010.